Serviceorientierte Infrastrukturen für vernetzte Dienste und eingebettete Geräte

Dissertation

zur Erlangung des akademischen Grades

Doktor-Ingenieur (Dr.-Ing.)

der Fakultät für Informatik und Elektrotechnik

der Universität Rostock

vorgelegt von
Bobek, Andreas, geb. am 05.06.1976 in Rostock
aus Rostock
Rostock, den 09.07.2008

urn:nbn:de:gbv:28-diss2009-0015-9

Gutachter:

- Prof. Dr.-Ing. Dirk Timmermann (Universität Rostock, Fakultät für Informatik und Elektrotechnik)
- Prof. Dr.-Ing. Hartmut Pfüller (Universität Rostock, Fakultät für Informatik und Elektrotechnik)
- Prof. Dr. rer. nat. Heiko Krumm (Universität Dortmund, Fakultät für Informatik)

Tag der Einreichung: 09.07.2008
Tag der Verteidigung: 12.12.2008

Danksagung

Die vorliegende Arbeit entstand während meiner Tätigkeit am Institut für Angewandte Mikroelektronik und Datentechnik an der Fakultät für Informatik und Elektrotechnik der Universität Rostock. Ich widme sie meinen Eltern, die mir eine hervorragende Ausbildung ermöglicht und mich seit jeher bei meinen Zielen und Wünschen unterstützt haben.

Ich möchte mich an dieser Stelle ausdrücklich bei Prof. D. Timmermann bedanken, der mir die Beschäftigung am Institut ermöglichte, sowie bei Dr. F. Golatowski, der mir stets den von mir benötigten Freiraum zur Verfügung gestellt hat. Beide haben Unglaubliches geleistet, damit diese Arbeit zustande kommen konnte.

Ein großer Dank geht an meine liebe Mutter Gisela Bobek und meinen Bruder Steffen Bobek, die dazu beigetragen haben, diese Arbeit leserlicher zu gestalten.

Weiterhin möchte ich mich bei meinem Kollegen Hendrik Bohn bedanken, mit dem ich viele fachliche Diskussionen und Gespräche, auch über das Fachgebiet hinaus, geführt habe. Mit seiner stets erfrischenden Laune hat er sehr zu einer angenehmen Arbeitsatmosphäre beigetragen.

Ein besonderer Dank geht an meinen Kollegen Elmar Zeeb, der in zahlreichen Diskussionen durch seine hohe Fachkompetenz und Sichtweise zu neuen Erkenntnissen verhalf. Viele Ergebnisse der vorliegenden Arbeit sind mit seiner Hilfe entstanden.

Für die Übernahme der Zweitgutachten möchte ich mich bei Prof. H. Pfüller und Prof. H. Krumm bedanken.

Zu guter Letzt bedanke ich mich bei allen Kollegen des Instituts für Angewandte Mikroelektronik und Datentechnik für das durchweg freundliche und angenehme Arbeitsklima.

Kurzfassung

Durch die Entwicklung der Informationstechnologie im Bereich der Hardware und der allgegenwärtigen Zugriffstechnologien wie WLAN oder UMTS sind die Voraussetzungen gegeben, um ubiquitäres Computing zu verwirklichen. Um jedoch eine Vielzahl von eingebetteten Systemen, Geräten und Diensten unterschiedlicher Anwendungsgebiete zu integrieren, bedarf es universell einsetzbarer bzw. domänenübergreifender Frameworks, die frei von Hardware-, Plattform- und Programmiersprachenabhängigkeiten sind, sowie Methodiken und Paradigmen, die die zunehmende Softwarekomplexität beherrschbar machen.

Die vorliegende Arbeit beschreibt deren wichtigste Anforderungen und stellt einige Frameworks vor. Für das Devices Profile for Web Services (DPWS) wird die vom Autor realisierte Umsetzung für den auf Java basierenden SOAP-Prozessor Axis2 vorgestellt. Mit Hilfe von DPWS wird anhand eines Szenarios demonstriert, wie das Paradigma der serviceorientierten Architekturen beim langfristigen Aufbau einer IT-Infrastruktur helfen kann. Als Anwendungsdomäne werden heterogene Lokalisierungssysteme als Dienste abgebildet und durch einen Lokalisierungsdienst verwendet.

Von der Erkenntnis ausgehend, dass das World Wide Web (WWW) mit seinen etablierten Standards wie URI, DNS und HTTP selbst eine universelle Plattform darstellt, wird mit der Web-oriented Device Architecture (WODA) ein eigener, alternativer Ansatz für ein Integrationsframework präsentiert. Für das REST-konforme WODA werden dazu Lösungen entwickelt, die die Grundprinzipien einer serviceorientierten Gerätearchitektur umsetzen.

Die Pipes-Plattform ist ein selbst entwickeltes, grafisches Modellierungswerkzeug, mit dessen Hilfe Anwendungen bzw. Prozesse erstellt werden können, die Geräte und Dienste verwenden oder neue Dienste erzeugen. Der Vorteil des Ansatzes besteht in der frameworkübergreifenden Verwendung von Diensten bzw. im domänenübergreifenden Einsatz durch die Verwendung der OSGi-Plattform. Das Modellierungswerkzeug kann in einem herkömmlichen Webbrowser verwendet werden, wodurch eine hohe Akzeptanz bzw. Erreichbarkeit gewährleistet ist.

Abstract

Through current advancements in embedded hardware and pervasive wireless access technologies such as WLAN or UMTS, the requirements for ubiquitious computing are already established. However, universal applicable and domain-spanning frameworks, which are independent of hardware platforms, operating systems and programming languages are required in order to integrate many embedded systems, devices and services coming from different fields of applications. Furthermore, paradigms and methodologies are essential to manage the increasing software complexity.

This thesis provides a survey of important requirements for such frameworks and introduces some of them. It shows the implementation of the Devices Profile for Web Services (DPWS) for Axis2 – a SOAP engine based on Java. The application of DPWS to an examplary scenario demonstrates how the Service-oriented Architectures (SOA) paradigm may help to evolve and maintain a flexible IT infrastructure over a long term. For this scenario heterogenous localization systems are represented as services and are used by a location service.

The Web-oriented Device Architecture (WODA) as a contribution of this thesis realizes an alternative approach for a suitable framework based on the World Wide Web (WWW). WODA focuses on Web protocols that are already in place and takes advantage of the fact that the WWW itself is an universal platform.

The Pipes platform allows to create new applications or processes that utilize devices and services. Pipes is part of this research and contains a browser-based modeling tool. Since the platform builds on OSGi, the approach may be applied for multiple frameworks and can be deployed in different domains.

Inhaltsverzeichnis

Abkürzungsverzeichnis xiii

Abbildungsverzeichnis xvii

Tabellenverzeichnis xxi

Listings xxiii

1 Einleitung 1

1.1 Eingebettete Systeme – Trends und Entwicklungen . . . 1
1.2 Integrationssoftware und Frameworks . . . 5
1.3 Anwendungsbeispiel: BlueTrack . . . 6
1.3.1 Architektur . . . 6
1.3.2 Probleme . . . 7
1.4 Herausforderung in der Geräte- und Dienstnutzung . . . 9
1.5 Schwerpunkte der Dissertation . . . 10
1.5.1 Abgrenzungskriterien . . . 11
1.5.2 Kapitelaufteilung . . . 11

2 Grundlagen 13

2.1 Das World Wide Web (WWW) . . . 13
2.1.1 URI . . . 13
2.1.2 HTTP . . . 13
2.1.3 Datenformate im WWW . . . 14
2.2 Serviceorientierte Architekturen . . . 15
2.2.1 Architektur und Paradigma . . . 15
2.2.2 SOA als evolutionäres Produkt . . . 18
2.3 Web Services . . . 21
2.3.1 Nachrichtenaustausch mit SOAP . . . 23
2.3.2 Beschreibung von Web Services mit WSDL . . . 25
2.3.3 UDDI als öffentlicher Verzeichnisdienst . . . 27

2.3.4 Das Basic Profile . . . 28
2.3.5 Adressierung mit WS-Addressing . . . 28
2.3.6 WS-Discovery als dynamischer Verzeichnisdienst . . . 29
2.3.7 Ereignisse mit WS-Eventing . . . 32
2.4 Representational State Transfer . . . 33
2.5 Zeroconf . . . 37
2.6 Web Application Description Language . . . 38
2.7 Universal Plug and Play . . . 38
2.8 Devices Profile for Web Services . . . 41

3 Stand der Wissenschaft und Technik 45
3.1 Bussysteme und Frameworks . . . 45
3.2 Middleware für Lokalisierungsdienste . . . 50
3.3 Grafische Prozess-Modellierungswerkzeuge für Daten, Dienste und Geräte . . . 52

4 Anforderungen an universelle, geräteintegrierende Frameworks 55
4.1 Allgemeine Anforderungen . . . 55
4.1.1 Abstraktion . . . 55
4.1.2 Geräte- und Dienstevielfalt . . . 55
4.1.3 Systemeigenschaften . . . 56
4.2 Basisdienste, Komponenten und Werkzeuge . . . 56
4.2.1 Plug-and-Play . . . 56
4.2.2 Beschreibung . . . 57
4.2.3 Ereignisse . . . 58
4.2.4 Sicherheit . . . 59
4.2.5 Software-Werkzeuge . . . 60
4.3 Anforderungen an das Netzwerk und die Kommunikation . . . 62
4.3.1 Netzwerk . . . 62
4.3.2 Kommunikation . . . 62
4.4 Fazit . . . 63

5 Umsetzung einer serviceorientierten Architektur mit dem Devices Profile for Web Services 65
5.1 DPWS-Implementierung für Apache Axis2 . . . 66
5.1.1 Apache Axis2 . . . 66
5.1.2 UDP als Transportprotokoll für SOAP . . . 69

5.1.3 Die Discovery-Erweiterung . . . 72
5.1.4 Die Eventing-Erweiterung . . . 75
5.1.5 Die DPWS-Erweiterung . . . 78
5.2 Geräte- und Dienstvorlagen für DPWS . . . 80
5.2.1 Motivation und Anforderungen . . . 80
5.2.2 Grundidee . . . 81
5.2.3 Typisierung von Diensten . . . 82
5.2.4 Typisierung von Geräten . . . 84
5.3 Klassifizierung von SOA-Teilnehmern . . . 86
5.4 Serviceorientierter Aufbau einer Lokalisierungsplattform mit DPWS . . . 87
5.4.1 Phase I: BlueScan . . . 88
5.4.2 Phase II: BlueScan2 – Eine neue BlueScan-Generation . . . 94
5.4.3 Phase III: Bewegungsprofile und Anpassung des BlueScan-Servers . . . 99
5.4.4 Phase IV: Discovery-Proxy . . . 102
5.4.5 Phase V: Ubisense – Ein neues Lokalisierungssystem . . . 103
5.4.6 Phase VI: Integration heterogener Lokalisierungssysteme . . . 105
5.4.7 Phase VII+: Zusammenfassung und Ausblick . . . 114
5.5 Implementierungen . . . 117
5.5.1 BlueScan-Geräte . . . 117
5.5.2 BlueScan-Server . . . 117
5.5.3 Ubisense-Gateway . . . 118
5.5.4 Discovery-Proxy . . . 118
5.5.5 Location-Service, Location-Manager und Object-Manager . . . 118
5.6 Zusammenfassung . . . 119

6 Die Web-oriented Device Architecture 121
6.1 Einführung in WODA . . . 122
6.1.1 Architektonische Grundlage . . . 122
6.1.2 Die sechs Phasen in WODA . . . 123
6.2 Adressierung (Addressing/Naming) . . . 124
6.3 Ankündigung und Entdeckung (Discovery) . . . 126
6.4 Beschreibung (Description) . . . 129
6.4.1 Dienstbeschreibung . . . 129
6.4.2 Gerätebeschreibung . . . 130
6.5 Steuerung (Control) . . . 131

6.6 Ereignisse (Eventing) . . . 132
6.7 Präsentation (Presentation) . . . 138
6.8 Anwendungsbeispiel . . . 140
6.9 Vergleich . . . 142
6.10 Zusammenfassung . . . 148

7 Die Pipes-Plattform 149
7.1 Anforderungen . . . 149
7.2 Grundidee und Konzept . . . 150
7.2.1 Architektur . . . 152
7.2.2 Ausführung . . . 154
7.3 Umsetzung . . . 155
7.3.1 Architektur der Plattformsoftware . . . 155
7.3.2 Pipes-Modeler . . . 156
7.3.3 Plug-in von Modulen . . . 158
7.3.4 Die Pipes-Engine . . . 162
7.3.5 Modul-Design in der Praxis . . . 163
7.3.6 Subpipes . . . 164
7.4 Anwendungsbeispiel . . . 164
7.5 Zusammenfassung . . . 167

8 Zusammenfassung und Ausblick 169
8.1 Zusammenfassung . . . 169
8.2 Ausblick . . . 173

Literaturverzeichnis 175

A Anhang – XML-Schemas für DPWS Geräte- und Diensttypen 189

B Anhang – Spezifikationen der DPWS-Lokalisierungssysteme und -dienste 191
B.1 Typdokumente der Lokalisierungssysteme . . . 191
B.2 WSDL-Dokumente der Lokalisierungssysteme und -dienste . . . 193

C Anhang – Spezifikationen der WODA-Dienste 205

D Anhang – Pipes 211

Abkürzungsverzeichnis

AoA	Angle Of Arrival
ARP	Address Resolution Protocol
ASN.1	Abstract Syntax Notation One
BPEL	Business Process Execution Language
CAN	Controller Area Network
CDC/CLDC	Connected Limited Device Configuration
CORBA	Common Object Request Broker Architecture
CSV	Comma Separated Value
DCOM	Distributed Component Object Model
DCP	Device Control Protocol
DCS	Distributed Control System
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
DNS-LLQ	DNS long-lived queries
DNS-SD	DNS Service Discovery
DPWS	Devices Profile for Web Services
EIB	Europäischer Installationsbus
EJB	Enterprise Java Beans
EPR	Endpoint Reference
ERM	Entity Relationship Model
ERP	Enterprise Resource Planing
ESB	Enterprise Service Bus
GENA	Generic Event Notification Architecture
GML	Geography Markup Language
HAVi	Home Audio Video Interoperability
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol

IDL	Interface Definition Language
IETF	Internet Engineering Task Force
IIOP	Internet Inter-ORB Protocol
IP	Internet Protocol
IPP	Internet Printing Protocol
IRI	Internationalized Resource Identifier
J2ME	Java Micro Edition
J2SE	Java Standard Edition
JMS	Java Message Service
JSON	JavaScript Object Notation
JSR	Java Specification Request
JTA	Java Transaction API
JVM	Java Virtual Machine
LIN	Local Interconnect Network
LON	Local Operating Network
mDNS	Multicast DNS
MEP	Message Exchange Pattern
MES	Manufacturing Execution System
MI	Message Information Header
MLP	Mobile Location Protocol
OASIS	Organization for the Advancement of Structured Information Standards
OMA	Open Mobile Alliance
OO	Objektorientierung
ORB	Object Request Broker
OSGi	Open Service Gateway Initiative
PDA	Personal Digital Assistant
PTR	DNS Record Type: Pointer
QName	qualified name
REST	Representational State Transfer
RF	Radio Frequency
RMI	Remote Method Invocation
RPC	Remote Procedure Call

SMTP	Simple Mail Transfer Protocol
SOA	Service-oriented Architectures
SOAP	Simple Object Access Protocol (veraltet)
SRV	DNS Record Type: Service
SSDP	Simple Service Discovery Protocol
TDoA	Time Differences Of Arrival
TLD	Top Level Domain
UBR	UDDI Business Registry
UDA	UPnP Device Architecture
UDDI	Universal Description, Discovery and Integration
UPnP	Universal Plug and Play
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
URN	Uniform Resource Name
UWB	Ultra Wide Band
W3C	World Wide Web Consortium
WADL	Web Application Description Language
WGS84	World Geodetic System 1984
WODA	Web-oriented Device Architecture
WS	Web Service
WS-*	Protokollfamilie der Web Services
WS-I	Web Services Interoperability Organization
WS4D	Web Services for Devices
WSDL	Web Services Description Language
WWW	World Wide Web
XML	Extensible Markup Language
XSLT	Extensible Stylesheet Language Transformation

Abbildungsverzeichnis

1.1 Trends, Entwicklungen und derzeitige Folgen in der Informationstechnologie bezogen auf Hardware und Geräte . . . 3
1.2 Eingebettete Geräte in den verschiedenen Anwendungsgebieten . . . 4
1.3 Architektur des BlueTrack-Systems . . . 7
1.4 BlueTrack-Installation in der Universität Rostock [12] . . . 8
1.5 Schwerpunkte der Dissertation . . . 10
1.6 Struktur der Arbeit mit Darstellung des eigenen Anteils . . . 11

2.1 Komponenten einer SOA . . . 17
2.2 Serviceorientierung als Folge langfristiger Entwicklungen in den Bereichen Programmiersprachen, verteilte Systeme und Geschäftsinformatik [27] . . . 18
2.3 Evolution der Komplexität von Softwaresystemen, Programmiersprachen und Paradigmen (Ausschnitt) . . . 19
2.4 Stufen der Softwareentwicklung, der wiederverwendbaren Elemente und der Beteiligten, die von der Wiederverwendung profitieren . . . 20
2.5 Web Services als Implementierung einer SOA . . . 22
2.6 Web Services Spezifikationen (Auszug) . . . 22
2.7 Struktur und Übertragung von SOAP-Dokumenten . . . 23
2.8 Struktur von WSDL-Dokumenten . . . 25
2.9 Komplexe Interaktionsmöglichkeiten für SOAP-Nachrichten durch die Verwendung von Message-Infomation-Header . . . 30
2.10 WS-Discovery Nachrichten in einem Netzwerk . . . 31
2.11 Nachrichten, Endpunkte und typischer Ablauf bei der Verwendung von WS-Eventing . . . 32
2.12 Ableitung des REST-Architekturstils [53] . . . 33
2.13 Abbildung einer Ressource auf eine Menge von Namen und Repräsentationen . . . 34
2.14 Der UPnP Protokollstapel . . . 39
2.15 Sequenzieller Ablauf der UPnP-Phasen . . . 40
2.16 Beziehungen zwischen Control-Point, Dienst, Aktionen und Zustandsvariablen . . . 40
2.17 DPWS-Protokollstapel . . . 42

2.18 Ablauf des DPWS-Discovery 43

3.1 Standardisierte OSGi-Services und ihre Domänen 48

4.1 Grundlegende Ereignismodelle in ereignisbasierten Anwendungen 59
4.2 Unterstützung der Lebenszyklus-Phasen der Dienste durch Software-Werkzeuge . 60

5.1 Gleichzeitige Verwendung von Axis2 als Dienst und Dienstnutzer 67
5.2 Fluss einer SOAP-Nachricht innerhalb der Axis2-Architektur 68
5.3 Nachrichtenfluss innerhalb des Discovery-Moduls 73
5.4 Funktionsweise des *Discovery Observers* 74
5.5 Sequenzdiagramm für Arbeitsweise des Eventing-Moduls 77
5.6 Nachrichtenfluss im Eventing-Modul für vorgegebene und benutzerspezifische Implementierung 78
5.7 Struktur der Dienstvorlagen 82
5.8 Struktur der Gerätevorlagen 84
5.9 Klassifizierung von SOA-Teilnehmern 86
5.10 Sequenzdiagramm für die Konfiguration, Inquiry und Ergebnisübertragung auf einem BlueScan-Gerät 89
5.11 Entity-Relationship-Modell für den BlueScan-Server 91
5.12 SOA-Diagramm für BlueScan-Geräte und BlueScan-Server 94
5.13 Sequenzdiagramm für die Konfiguration, Inquiry und Ereignisse eines BlueScan2-Gerätes 95
5.14 SOA-Diagramm für BlueScan-Geräte der Versionen 1 und 2 und BlueScan-Server 100
5.15 SOA-Diagramm nach Abschluss der 3. Phase 102
5.16 Discovery über Netzwerkgrenzen hinweg mittels eines Discovery-Proxies 103
5.17 SOA-Diagramm für die Ubisense-Integration 106
5.18 XML-Struktur der Parameter des Lokalisierungsdienstes 109
5.19 Dienst-Sequenzdiagramm zur Lokalisierung eines Mitarbeiters 110
5.20 Dienst-Sequenzdiagramm zur Lokalisierung eines Mitarbeiters mit Angabe eines Qualifizierers ($p_{incl} = 100\%$) 112
5.21 Dienst-Sequenzdiagramm zur Lokalisierung eines Mitarbeiters mit Angabe eines Qualifizierers ($p_{qual} = \textit{best quality}$) (Ausschnitt) 114
5.22 SOA-Diagramm für die Integration heterogener Lokalisierungssysteme 115
5.23 Einfache Erweiterbarkeit durch Integration neuer Dienste 116

6.1 UPnP und DPWS: Für Geräte mit browserbasierter Steuerung sind zwei unterschiedliche Schnittstellen notwendig . . . 121
6.2 WODA-Komponenten, Ressourcen und Links . . . 123
6.3 Phasen für die Integration und Nutzung von UPnP-, DPWS- und WODA-Diensten bzw. -Geräten . . . 125
6.4 Der WODA-Stack . . . 126
6.5 Auflisten von typisierten Dienstinstanzen innerhalb bestimmter Domains mit Zeroconf [54] . . . 127
6.6 Wurzelressourcen dreier Dienste und gemeinsame Ressource für die Gerätebeschreibung . . . 131
6.7 Eventing-Stack von WODA . . . 132
6.8 Änderungen von Ressourcen infolge eines Ereignisses . . . 133
6.9 Abonnieren und Veröffentlichen von Ressourcen . . . 134
6.10 Beenden von Abonnements . . . 135
6.11 Beispiel für eine Eventing-Extension: Verwaltung von Abonnements . . . 136
6.12 Verwendung einer *Resource Factory* für benutzerdefinierte, abonnierbare Ressourcen . . . 137
6.13 Auslagerung der Abonnementverwaltung und Nachrichtenveröffentlichung auf eine externe Verteilerkomponente . . . 138
6.14 Verwendung der *gleichen* Schnittstelle sowohl für die Maschine-zu-Maschine- als auch für die Mensch-zu-Maschine-Kommunikation . . . 139
6.15 Ressourcen des Ubisense-Gerätes . . . 143
6.16 Abbildung von und Beziehungen zwischen Geräten und Diensten in UPnP, DPWS und WODA . . . 143

7.1 Modulares Konzept für die Umsetzung der Anforderungen an die Pipes-Plattform 151
7.2 Zentraler Steuerungsansatz für flexible Prozesse und Anwendungen unter Berücksichtigung der Anforderungen in der Pipes-Plattform . . . 152
7.3 Module, Ports, Kabel und Pipes . . . 153
7.4 Module und Pipes als OSGi-Bundles . . . 155
7.5 Technische (Modul-Entwickler) und semi-technische (Pipes-Entwickler) Benutzer der Pipes-Plattform . . . 156
7.6 Screenshot des Pipes-Modelers . . . 157
7.7 Typische Implementierung eines Moduls . . . 163
7.8 Pipes Beispielszenario . . . 165
7.9 Pipes Beispielszenario (Fortsetzung) . . . 166

Tabellenverzeichnis

5.1 WSDL-Operationen des BlueScan-Gerätes 88
5.2 WSDL-Operationen des BlueScan-Servers 91
5.3 Inhalt der Ereignisnachrichten 96
5.4 WSDL-Operationen des BlueScan2-Gerätes 97
5.5 WSDL-Operationen des Ubisense-Gateway-Dienstes 104

6.1 Ressourcen für den Ubisense-Dienst 141
6.2 Vergleich der Frameworks bezüglich Discovery 144
6.3 Vergleich der Frameworks bezüglich Ereignisse 146

Listings

2.1 Beispiel einer HTTP-Anfrage . . . 14
2.2 Beispiel einer HTTP-Antwort . . . 14
2.3 Beispiel einer SOAP-Nachricht [43] . . . 24
2.4 Beispiel einer WSDL-Service-Definition (Auszug) [46] . . . 26
2.5 Beispiel einer SOAP-Nachricht mit Message-Information-Headers [50] . . . 29

5.1 Konfiguration des *soapudp*-Moduls in der axis2.xml . . . 70
5.2 Konfiguration eines *Target Services* in der services.xml . . . 74
5.3 Konfiguration eines Gerätes in der services.xml . . . 78
5.4 Beispiel für eine Dienstvorlage . . . 83
5.5 Beispiel für eine Gerätevorlage . . . 85
5.6 Algorithmus auf dem BlueScan-Gerät (informal) . . . 88
5.7 BlueScan-ScanService-Typ . . . 90
5.8 BlueScan-Device-Typ . . . 90
5.9 Algorithmus der Scan-Operation auf dem BlueScan-Server (informal) . . . 92
5.10 Algorithmus der Find-Operation auf dem BlueScan-Server (informal) . . . 92
5.11 Algorithmus der Registrierung von BlueScan-Geräten (informal) . . . 93
5.12 Algorithmus zum Empfangen von ScanResult-Ereignissen (informal) . . . 93
5.13 XML-Schema-Definition für den Konfigurationstyp . . . 96
5.14 Beispiel für eine Konfiguration . . . 97
5.15 BlueScan-Device-Typ Version 2 . . . 97
5.16 Algorithmus des Inquiry-Threads auf dem BlueScan-Gerät 2 (informal) . . . 98
5.17 Algorithmus des BlueScan-Gerätes Version 2 (informal) . . . 99
5.18 Algorithmus der Registrierung von BlueScan-Geräten 1 und 2 (informal) . . . 101
5.19 Algorithmus beim Empfangen von ScanEvent-Ereignissen (informal) . . . 101
5.20 Anwendung des *BoxFilters* beim Senden einer *Subscribe*-Nachricht an das Ubisense-Gateway . . . 105

6.1 Ausschnitt aus einem WADL-Dokument . . . 130
6.2 WODA-Gerätebeschreibung für das Ubisense-Gerät . . . 142

7.1 Auszug aus einer Modul-JavaScript-Datei am Beispiel des JSON-Builders 159
7.2 Auszug aus dem zum Screenshot zugehörigen Pipes-Dokument 162

A.1 XML-Schema für Dienst- und Gerätetypisierung 189

B.1 ScanService-Diensttyp 191
B.2 EventService-Diensttyp 191
B.3 BlueScan-Gerätetyp 191
B.4 BlueScan-Gerätetyp der 2. Generation 192
B.5 BlueScan-ScanService-WSDL 193
B.6 BlueScan-EventService-WSDL 194
B.7 BlueScan-Server-WSDL 195
B.8 Ubisense-Gateway-WSDL 198
B.9 Location-Service-WSDL 200

C.1 Relevanter Auszug aus dem XML-Schema von DPWS für WODA-Geräte 205
C.2 XML-Schema für Beschreibung von WODA-Geräten 206
C.3 XML-Schema für den WODA-Ubisense-Dienst 207
C.4 WADL-Dokument für den WODA-Ubisense-Dienst 208
C.5 WODA-Beschreibung für das Ubisense-Gerät 210

D.1 Beispiel für ein Pipes-Dokument 211

1 Einleitung

1.1 Eingebettete Systeme – Trends und Entwicklungen

Die berühmte Schätzung des IBM-Chefs Thomas Watson aus dem Jahr 1943, laut welcher er den Bedarf an Computern für die US-Wirtschaft auf höchstens fünf Stück bezifferte [1] (und die in den 50er Jahren durch die Firma Arthur D. Little auf 50 Stück erhöht wurde) sowie das Zitat Ken Olsons, Präsident, Vorsitzender und Gründer von Digital Equipment Corp. aus dem Jahr 1977: „Es gibt keinen Grund, warum irgendjemand einen Computer in seinem Haus wollen würde." (alles [2]), zeigen, wie schwer es selbst für Fachleute ist, einen Markt (Bedarf) vorherzusagen, der einige Jahre später die Welt veränderte und heute als *Informationstechnologie* zum fünften und jüngsten Kondratieff-Zyklus zählt [3].

Die Entwicklung der Informationstechnologie unterliegt seitdem drastischen Veränderungen, die auch gegenwärtig und zukünftig anhalten werden. Im Bereich der Hardware sind dieses im Einzelnen:

- **Verringerung des Formfaktors bzw. Erhöhung der Informationsdichte** Während die ersten größeren Rechenmaschinen noch ganze Räume füllten, führte die Erfindung von integrierten Schaltkreisen zu immer kleineren Chips. Die durch Gordon Moore im Jahr 1965 vorhergesagte Entwicklung, dass sich die Transistorendichte etwa alle 18 bis 24 Monate verdoppelt, gilt auch weiterhin [4] (Moore'sches Gesetz). Mikrocontroller für den Einsatz in Sensornetzwerken sind inzwischen im Millimeterbereich angekommen, siehe z. B. das *Smart Dust Projekt* [5].
- **Steigende Leistungsfähigkeit bzw. Taktfrequenz** Der 1952 gebaute Z5 wurde noch mit 50 Hz betrieben, das IBM-Modell 5150 (1981), welches zum ersten „Heimcomputer" zählt, bereits mit 4,77 MHz. Moderne Prozessoren erlauben Taktfrequenzen von mehreren GHz und selbst in Mobiltelefonen sind aktuell 500 MHz-Prozessoren zu finden.
- **Fallende Kosten** Die ersten Computer konnten sich nur Regierungen oder große Firmen leisten. Die Kosten lagen, auf heutige Kaufkraft bezogen, bei beispielsweise mehreren Millionen Euro[1] (UNIVAC I, 1951) oder 150.000 EUR (Z5, 1952). Aktuelle komplett ausgestattete Bürocomputer sind bereits für 1.000 EUR, eingebettete Systemplattformen wie

[1] ca. 5,3 Mio. EUR bei einer angenommenen Inflation von 3% p. a. und 1,5 Mio. US-Dollar damaliger Kosten

das FOX Board [6] für etwas über 100 EUR zu bekommen. Ziel ist es, vollständige Systeme im Centbereich anbieten zu können.

- **Sinkender Stromverbrauch** Neben den aufgeführten Fixkosten müssen auch variable Kosten berücksichtigt werden, die vor allem durch den Stromverbrauch verursacht werden. Die Leistungsaufnahme des ENIACs (1946) betrug 140 kW, die des Z5 6 kW. Bei einem heutigen angenommenen Strompreis von 15 Cent/kWh, würde der Betrieb mit ca. 500 EUR bzw. 22 EUR täglich zu Buche schlagen. Moderne Laptops kommen mit 40 W (ca. 15 Cent) aus, Sensorelektronik wie die oben erwähnten Smart-Dust-Mikrocontroller liegen im Milli- bis Nanowattbereich.
- **Physikalische Schnittstellen** Die ersten Rechenmaschinen besaßen überhaupt keine physikalischen Schnittstellen. Auch bei den nächsten Generationen waren sie hauptsächlich auf das Anschließen von externen Geräten (Tastatur, Floppylaufwerk usw.) beschränkt und vor allem immer systemspezifisch. Mit der Vernetzung von Computern wurde der Bedarf geschaffen, Schnittstellen zu standardisieren. Zu den gängigen systemübergreifend eingesetzten Standards gehören heute USB, Firewire, Ethernet, WLAN, Bluetooth usw.

Die aufgeführten Trends führen derzeit im Wesentlichen zu drei Folgeerscheinungen:

- **Mobilität** Geräte, die ihren Ort mehr oder weniger häufig ändern, werden als *mobil* bezeichnet. Ihre Zahl nimmt ständig zu. Allein in Deutschland betrug die Anzahl der Mobiltelefonanschlüsse im April 2008 über 100 Millionen. Neben dem Mobiltelefon als bekanntesten Vertreter zählen aber auch Kraftfahrzeuge oder zukünftig Sensorelektronik in *intelligenter Kleidung* (*wearable computing*) [7] usw. zu mobilen Geräten. Für das *Mobile Computing* sind zwei weitere Trends verantwortlich: Durch die ständige Verkleinerung der Bauteile können Gewichte erzielt werden, die auch wirklich *tragbar* sind. Schließlich werden die meisten mobilen Geräte erst durch Menschen mobil. Weiterhin findet fortwährend der Ausbau der Infrastruktur im Telekommunikationsbereich statt, wodurch sich Zugriffstechnologien wie WLAN, UMTS oder Glasfasernetze bewähren.
- **Verschmelzung von Geräten/Funktionalitäten** Bedingt durch die Miniaturisierung und kostengünstige Herstellung vieler Bauteile finden immer mehr unterschiedliche Hardwarekomponenten Platz in einem einzigen Gerät. Sehr deutlich wird das bei Mobiltelefonen, in denen Sensoren (Helligkeit, Neigung), Foto-, Videokamera, Leselampe sowie diverse physikalische Schnittstellen zunehmend zur Standardausstattung gehören.
- **Ubiquitäres Computing** Fast alle o. g. Faktoren ermöglichen die (beginnende) Umsetzung des *Ubiquitous Computing*, dessen Begriff bereits 1988 von Mark Weiser geprägt wurde [8] und welcher die Vision der allgegenwärtigen, nicht mehr wahrnehmbaren Com-

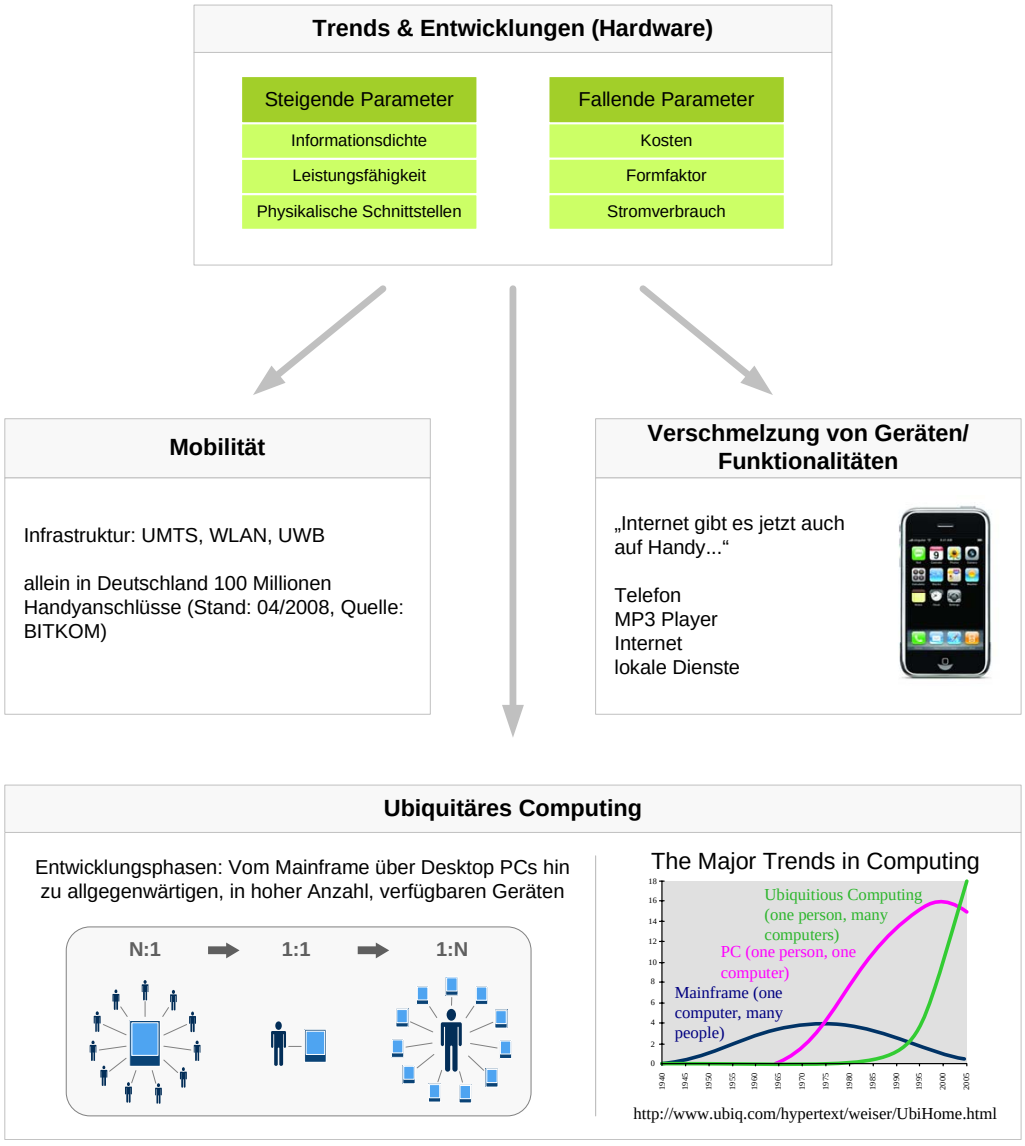

Abbildung 1.1: Trends, Entwicklungen und derzeitige Folgen in der Informationstechnologie bezogen auf Hardware und Geräte

puter, die in hoher Anzahl in die Umgebung (Welt) integriert sind und den Menschen dienen, beinhaltet. Diese Phase ist gekennzeichnet durch die Ablösung der Mainframe- und PC-Ära (Abbildung 1.1), siehe auch [9].

Abbildung 1.2: Eingebettete Geräte in den verschiedenen Anwendungsgebieten

Geräte, die aus Hard- und Softwaresicht eigenständige Computer darstellen, werden oft auch als *eingebettete Geräte* oder *eingebettete Systeme* bezeichnet. Sie sind in der Regel dadurch gekennzeichnet, dass sie nur eine bestimmte Aufgabe innerhalb eines größeren Systems übernehmen und nur in dessen Kontext funktionieren. Beispiele dafür sind Regelsysteme in Kraftfahrzeugen, Steuerungsmodule in chemischen Anlagen, Personal Digital Assistants (PDA), Mobiltelefone, Unterhaltungselektronik, Haushaltsgeräte usw.

Durch die genannten Trends und Entwicklungen können zunehmend neue Anwendungsgebiete (*Domänen*) für den Einsatz von eingebetteten Systemen erschlossen werden. Gleichzeitig nimmt die Vielfalt dieser Geräte ständig zu (Abbildung 1.2). Typische Domänen sind die Automatisierungstechnik (Industrie- und Fertigungsstraßen), die Telekommunikationsbranche, die Automobilindustrie sowie Gebäudeautomatisierung. Neu hinzugekommen sind die Heimautomatisierung, soziale Anwendungsgebiete (z. B. [10]) oder *Ubiquitous Games* [11].

Mit dem Fortschreiten der Allgegenwärtigkeit eingebetteter Systeme steigt auch der Bedarf, diese in unterschiedlichen Kontexten zu nutzen. Die Interaktion soll nicht länger auf eine be-

stimmte Domäne beschränkt sein, sondern vielmehr *domänenübergreifend* erfolgen können. Beispielsweise erfordern neue Gesetze in der Lebensmittelindustrie eine lückenlose Überwachung und Einhaltung der Kühlkette bei bestimmten Produkten. Hier greifen diverse Systeme wie Temperatursensoren, Protokollierung, Fahrzeugüberwachung, Flottenmanagement, RFID- oder Barcodetechnik aus dem Logistikbereich usw. ineinander. Daran sind unterschiedliche (u. a. mobile) Geräte, Softwarekomponenten und Menschen beteiligt. Ziel sollte es sein, dass dem Endkunden auch Tage nach dem Einkauf die gewünschten Informationen zum Produkt zur Verfügung stehen, etwa inwiefern die Kühlkette tatsächlich eingehalten wurde oder ein Mindesthaltbarkeitsdatum, welches sich daraus dynamisch errechnen ließe. Derartige Auskünfte sind derzeit noch nicht möglich bzw. nicht realisiert.

1.2 Integrationssoftware und Frameworks

Alle bis hier genannten Argumente und Überlegungen betreffen die Geräte und die Domänen, in denen sie vorkommen. Gleichzeitig führen sie aber auch zu einer bisher noch nicht erwähnten Herausforderung: Die Komplexität der Software nimmt ständig zu. Software für Geräte lässt sich in zwei Arten unterteilen: Die *Steuerungssoftware* ist zuständig für die Umsetzung der eigentlichen Gerätefunktionalitäten. Sie ist größtenteils abhängig vom Gerätetyp und überwacht beispielsweise physikalische Größen oder wertet die gedrückten Tasten auf einer Fernbedienung aus. Die *Integrationssoftware* dagegen ermöglicht die Integration des Gerätes in eine bestimmte Umgebung – in der Regel ein Netzwerk – sowie die Nutzung des Gerätes durch andere. Diese Art Software muss auf gemeinsamen Protokollen beruhen, die die Integrationssoftware anderer Geräte ebenfalls verstehen. Wenn in dieser Arbeit von *Geräteintegration* die Rede ist, dann wird damit die Integrationssoftware für Geräte bzw. auf Geräten adressiert. Die Protokolle, Methoden, Werkzeuge usw., die die gemeinsame Grundlage der Integration bilden, sollen im Weiteren mit *Integrationsframework* oder einfach nur *Framework* bezeichnet werden.

Folgende Besonderheiten sind kennzeichnend für Integrationssoftware bzw. einem derartigen Integrationsframework:

- **Spontane Vernetzung** Unter *spontaner Vernetzung* wird die Möglichkeit verstanden, dass sich Geräte ohne manuelle Konfiguration selbstständig vernetzen können und sich untereinander entdecken bzw. registrieren, so dass anschließend eine Kommunikation (Datenaustausch) stattfinden kann. Diese Fähigkeit ist umso wichtiger, je dynamischer (mobiler) die Netzwerke/Geräte sind.
- **Kommunikation** Die Kommunikationsinfrastruktur ermöglicht den eigentlichen Datenaustausch zwischen den Geräten. Verschiedene Ausprägungen wie Punkt-zu-Punkt oder

Broadcast, synchrone und asynchrone Kommunikationsmuster existieren.

- **Ereignismechanismus** Um den Zustand eines Gerätes beobachten zu können, müssen für die Ereignisverteilung und -registrierung entsprechende Mechanismen zur Verfügung stehen.
- **Typisierung** Unterschiedliche Gerätetypen verlangen oftmals unterschiedliche Anwendungsprotokolle (auch: *Domänenprotokolle*). Je nachdem, wie hochgradig dynamisch eine bestimmte Umgebung ist, müssen die Gerätetypen identifiziert werden können. In der Regel geschieht das über Dokumente, die ein Gerät und dessen Dienste beschreiben.
- **Weitere gemeinsame Dienste** Dazu zählen höherwertige Dienste, mit denen Geräte beispielsweise verwaltet werden können oder Transaktions- und Sicherheitsdienste.

Der Wahl eines geeigneten Integrationsframeworks kommt zukünftig eine immer wichtigere Bedeutung zu, da die Grenzen der oben erwähnten Domänen zunehmend aufgeweicht werden, die Vielfalt der Geräte zunimmt und der Grad der Mobilität wächst. Um möglichst viele Geräte und Domänen einbeziehen und damit völlig neue Szenarien realisieren zu können, müssen Integrationsframeworks möglichst universell, das heißt domänenübergreifend anwendbar sein.

1.3 Anwendungsbeispiel: BlueTrack

Einige Beispiele und Diskussionen in der vorliegenden Arbeit beruhen auf einer existierenden Anwendung, in welcher mehrere Probleme demonstriert werden können, die in ihrer ursprünglichen Umsetzung aufgetaucht sind. Diese sind bezeichnend für viele Anwendungen ähnlicher Art.

1.3.1 Architektur

BlueTrack [12] ist ein auf Bluetooth [13] basierendes Trackingsystem, welches am Institut für Angewandte Mikroelektronik und Datentechnik (MD) an der Universität Rostock im Einsatz ist und mit welchem sich Bewegungsprofile von Bluetooth-Geräten wie Mobiltelefonen oder PDAs erstellen lassen. Dazu nutzt BlueTrack den Umstand, dass bei vielen Geräten Bluetooth aktiviert ist und diese automatisch und für den Benutzer unbemerkt auf Suchanfragen antworten.

Abbildung 1.3 zeigt den schematischen Aufbau des BlueTrack-Systems. Mehrere BlueTrack-Sensoren senden in regelmäßigen Abständen *Inquiry*-Nachrichten aus, die von Bluetooth dazu genutzt werden, um nach Geräten zu suchen. Alle Geräte, die sich in Reichweite der Nachricht befinden, senden eine Antwort, die u. a. die jeweilige 6 Byte lange Bluetooth-Device-Adresse enthält, die das antwortende Gerät eindeutig identifiziert (z. B. *00:60:57:39:E7:AE*).

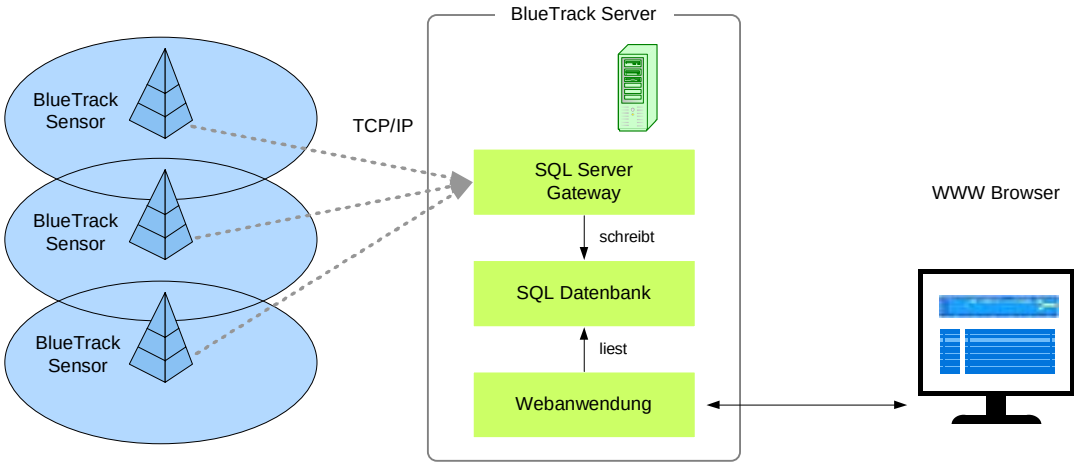

Abbildung 1.3: Architektur des BlueTrack-Systems

Die Daten werden an das SQL-Gateway des zentralen BlueTrack-Servers gesendet, ausgewertet und von dort aus als Datensätze (Bluetooth-Device-Adresse, Zeitpunkt in Reichweite, Zeitpunkt außerhalb Reichweite) in eine Datenbank geschrieben. Eine ebenfalls auf dem BlueTrack-Server befindliche Webanwendung bereitet die gesammelten Daten aller BlueTrack-Sensoren grafisch auf und kann sie in einem herkömmlichen Webbrowser präsentieren. Die Webanwendung zeigt u. a. vollständige Bewegungsprofile für ein beliebig gewähltes Gerät über einen beliebig langen Zeitraum.

Die Qualität der gewonnenen Profile ist vor allem von der Anzahl der installierten Sensoren sowie von deren Aufstellungsorten abhängig (Abbildung 1.4). Als Mindestvoraussetzung gilt, dass jeder zu beobachtende Ort von mindestens einem Sensor erreicht werden kann. Überlappende Bereiche erhöhen die Genauigkeit der Messungen. Je kleiner die Überlappungsbereiche sind, desto genauere Bewegungsprofile lassen sich erzielen.

Die Sensoren wurden als eingebettete Systeme auf ca. 7x7 cm großen FOX Boards implementiert [6]. Auf diesen laufen ein Linux-System und der BlueZ-Stack [14], mit welchem die Inquiries durchgeführt werden. Die Sendeleistung wurde mit der Verwendung eines Bluetooth-Senders der Sendeleistungsklasse 3 beschränkt. Diese (kleinste) spezifizierte Klasse erlaubt Übertragungen im Meterbereich und ist für einzelne Räume geeignet [15].

1.3.2 Probleme

Das EU-Projekt *Lokale Mobile Dienste* (*Local Mobile Services*, LOMS) [16], an welchem sich auch das Institut MD beteiligt, beschäftigt sich mit der Erzeugung, Bereitstellung und dem

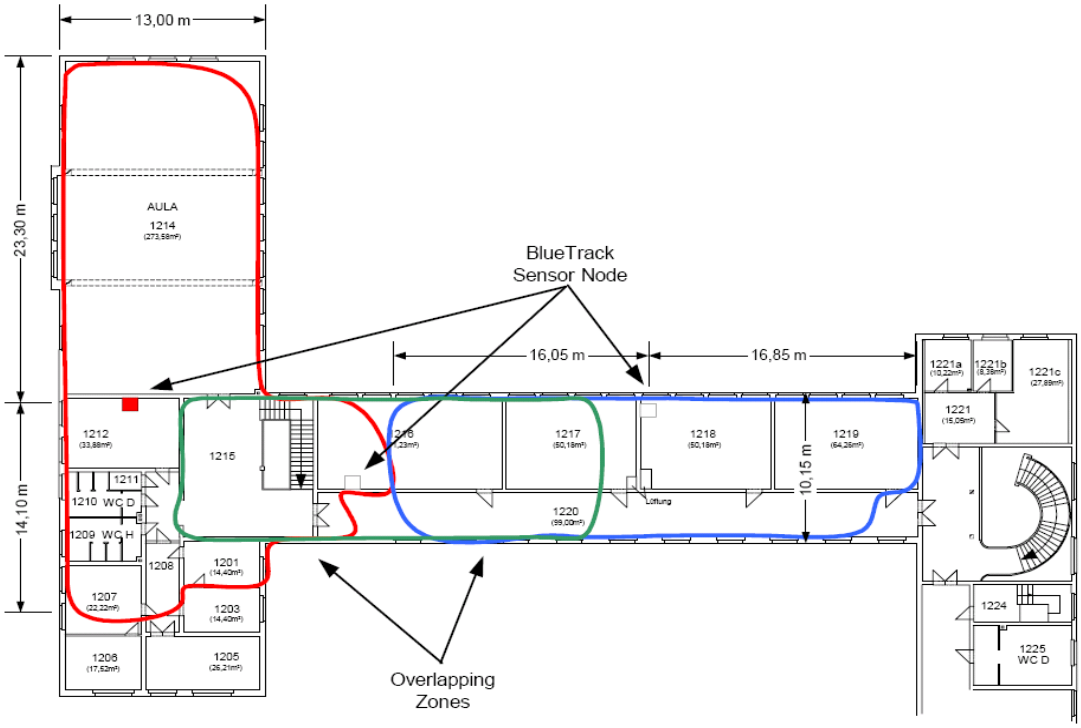

Abbildung 1.4: BlueTrack-Installation in der Universität Rostock [12]

Betrieb lokaler mobiler Dienste. Die Lokalisierbarkeit des Nutzers ist Grundlage für das Anbieten ortsabhängiger Dienste. Im Rahmen eines Demonstrators sollte das BlueTrack-System als ein Lokalisierungssystem verwendet werden. Bisher war es jedoch nur für das Tracken von Geräten (das Erstellen von Bewegungsprofilen) konzipiert.

Im Zusammenhang mit der Erweiterung des BlueTrack-Systems vom Tracken hin zum Lokalisieren konnten folgende Probleme identifiziert werden:

- **Monolithische Umsetzung bzw. starre Kopplung zwischen Sensoren und Server** Die BlueTrack-Sensoren generieren aus den gesammelten Daten direkt SQL-Anweisungen in der Form *INSERT INTO ...*. Dadurch besteht eine starke Abhängigkeit zwischen Sensor und dem Datenbankschema auf dem Server. Weiterhin ist die IP-Adresse des SQL-Gateways fest in die Anwendung einprogrammiert worden. Nach einer Änderung dieser Adresse funktioniert das gesamte System erst dann wieder, wenn *alle* Sensoren mit der entsprechenden Programmänderung bespielt wurden.
- **Erweiterbarkeit des Systems** Die Erweiterbarkeit des Systems ist aufgrund der zu-

vor genannten Punkte stark beschränkt und grundsätzlich nur manuell und durch hohen Aufwand realisierbar. Änderungen an der BlueTrack-Server-Konfiguration (beispielsweise Änderung des Datenbankschemas, des Datenbanktyps oder der Serveradresse) ziehen auch immer Änderungen der Sensoren nach sich.

- **Beschränkte Funktionalitäten** Die Sensoren sind von außen nicht konfigurierbar oder steuerbar. Beispielsweise lässt sich das Intervall des Scanvorganges nicht einstellen. Sinnvoll wäre auch eine Art Ruhemodus, in welchen die Sensoren nachts gesetzt werden könnten, da in dieser Zeit keine zu trackenden Ziele zu erwarten sind.[2]
- **Anwendbarkeit in einem anderen Kontext** Um das BlueTrack-System in einem anderen Kontext anwenden zu können, müsste es flexibler entworfen sein. Beispielsweise lassen sich weder die gemessenen noch die ausgewerteten Daten von außen abfragen. Es existiert lediglich eine Schnittstelle (die der Webanwendung). Zur automatischen Weiterverarbeitung ist diese jedoch nicht geeignet. Auch lassen sich keine ereignisbasierten Anwendungen realisieren. Sinnvoll wäre eine Überwachung auf das Vorhandensein bestimmter registrierter Geräte in definierten Bereichen.

Die genannten Probleme sind typisch für viele Anwendungen und Systeme. Sie sind meistens durch unflexible und monolithische Architekturen gekennzeichnet. Die vorliegende Arbeit beschäftigt sich unter anderem mit der Frage, wie derartige Systeme flexibel und entkoppelt gestaltet werden können, so dass eine Wiederverwendung bzw. die Möglichkeit der Integration in andere Systeme/Anwendungen gegeben ist.

1.4 Herausforderung in der Geräte- und Dienstnutzung

Die bloße Existenz von Daten, Diensten und Geräten allein reicht nicht aus. Um höherwertige Ziele zu erreichen, neue Dienste zu erstellen und Anwendungen zu schaffen, die auf mobile Geräte und Dienste zugreifen, müssen Programme geschrieben oder Prozesse/Workflows definiert werden. Eine besondere Möglichkeit stellen grafische Werkzeuge dar, mit deren Hilfe sich solche Anwendungen modellieren lassen, da sie in der Regel auch von weniger technisch versierten Benutzern genutzt werden können und gleichzeitig einen besseren visuellen Einblick in das Ziel der Anwendung geben. Gerade in einem von Heterogenität gekennzeichneten Umfeld stellen grafische Werkzeuge eine Alternative dar, da sie die technologischen Komplexitäten verstecken und gleichzeitig einem breiteren Anwenderkreis die Benutzung erlauben, als dieses bei herkömmlicher Programmierung der Fall wäre.

[2] Das betrifft die Installation in einem öffentlichen Gebäude wie der Universität Rostock. Die Anforderungen an das gleiche System für einen anderen Ort könnten anders lauten.

1.5 Schwerpunkte der Dissertation

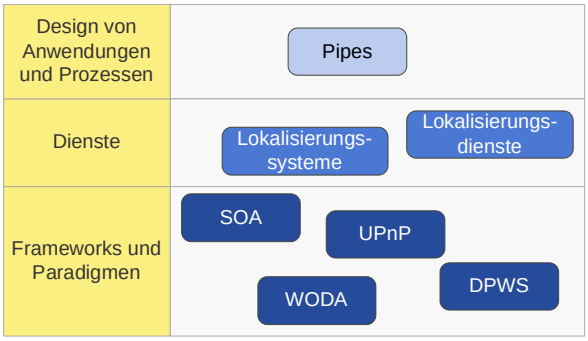

Abbildung 1.5: Schwerpunkte der Dissertation

Schwerpunkte dieser Dissertation sind Konzeption, Umsetzung sowie Anwendung von Frameworks, die Merkmale serviceorientierter Architekturen (SOA) aufweisen und Geräte integrieren, der SOA-basierte Aufbau einer Lokalisierungsplattform und die grafische Anwendungs- bzw. Prozessmodellierung mit der *Pipes-Plattform*. Es handelt sich dabei um eine vertikale Vorgehensweise, die die drei horizontalen Ebenen der Frameworks/Paradigmen, der Dienste sowie des Designs von Anwendungen und Prozessen vereint (Abbildung 1.5).

Bei den einzelnen Arbeiten steht die *universelle* Anwendbarkeit im Vordergrund. Die vorgestellten Ansätze stellen daher keine Lösungen für einzelne domänen- und kontextabhängige Probleme dar, sondern sollen vielmehr *domänenübergreifend* funktionieren und auf Heterogenität zurückzuführende Hürden überwinden. Im Einzelnen werden dazu folgende Punkte bearbeitet:

- **Anforderungen an universelle Frameworks** In Kapitel 4 werden informativ allgemeine Anforderungen aufgeführt, die universell einsetzbare Frameworks erfüllen sollten.
- **Devices Profile for Web Services (DPWS)** Es wird gezeigt wie sich mittels serviceorientiertem Entwurf und der Web-Services-Technologie Anwendungen und Geräte integrieren lassen. Dazu wird beispielhaft eine Lokalisierungsarchitektur vorgestellt, die verschiedene Lokalisierungssysteme vereint. Weiterhin wird ein eigener Ansatz vorgestellt, mit dem standardisierte bzw. typisierte Geräte und Dienste für DPWS erstellt werden können, und es wird die Umsetzung von DPWS für das Apache-Axis2-Projekt vorgestellt.
- **Web-oriented Device Architecture (WODA)** Kapitel 6 präsentiert einen eigenen alternativen Ansatz für ein Integrationsframework, welches u. a. etablierte Webtechnologien verwendet und daher dem Anspruch an Universalität gerecht wird.
- **Pipes** Die Pipes-Plattform ist ein vom Autor entwickeltes Werkzeug, mit welchem Geräte-

und Dienste-Mashups unterschiedlicher Frameworks erstellt, ausgeführt und überwacht werden können. In diesem Kapitel werden die Architektur der Plattform, die Umsetzung sowie ein Anwendungsbeispiel gezeigt. Die Pipes-Plattform ist ein universeller Ansatz, da sie frameworkübergreifend eingesetzt werden kann.

1.5.1 Abgrenzungskriterien

Die Arbeit beschäftigt sich ausschließlich mit Integrationssoftware, Diensten und Anwendungsdesign. Es werden keine Betrachtungen zu Hardware, Echtzeit oder speziellen Bussystemen angestellt.

1.5.2 Kapitelaufteilung

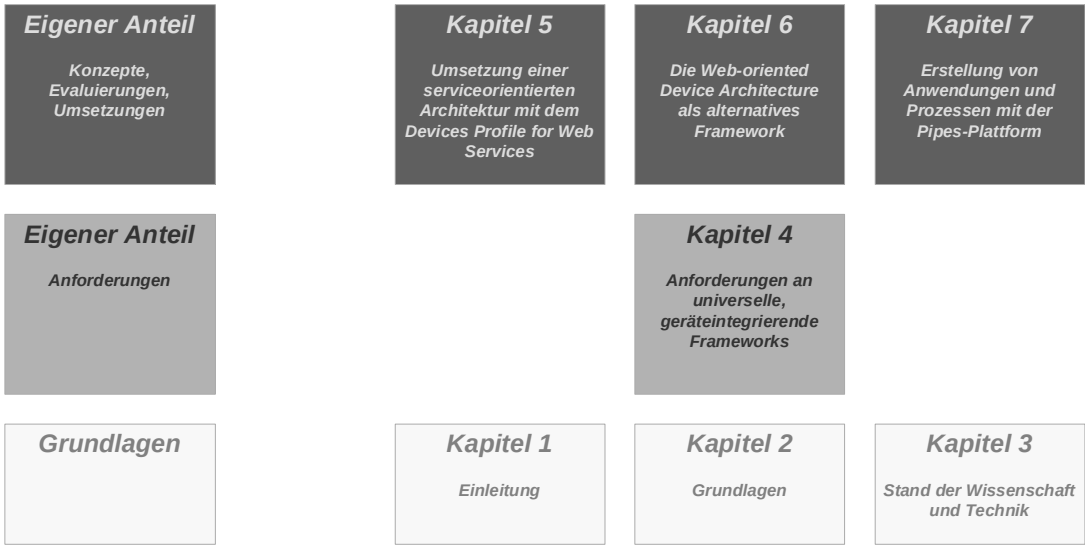

Abbildung 1.6: Struktur der Arbeit mit Darstellung des eigenen Anteils

Abbildung 1.6 zeigt die sieben Hauptkapitel der vorliegenden Arbeit. Die Kapitel 1–3 behandeln die Grundlagen. Kapitel 4–7 stellen mit den Anforderungen, Konzepten, Evaluierungen und Umsetzungen den eigenen Anteil dar.

2 Grundlagen

2.1 Das World Wide Web (WWW)

Das *World Wide Web* (WWW) wurde 1989 am CERN von Time Berners-Lee entwickelt und stellt heute die größte Anwendung im Internet dar [17]. Es wurde ursprünglich als reines Hypermedia-System zum transparenten Austausch von Daten entworfen, um die bestehenden Hürden zwischen den unterschiedlichen Plattformen und Dateisystemen zu überwinden.

Die drei Spezifikationen *Uniform Resource Identifier* (URI) [18], *Hypertext Transfer Protocol* (HTTP) [19] sowie *Hypertext Markup Language* (HTML) [19] bilden die technische Grundlage für das WWW.

2.1.1 URI

Eine URI ist ein Name bzw. ein Identifikator für eine Ressource, die eine über die Zeit konstante Semantik besitzt. Sie wird als String repräsentiert, welcher aus zwei Teilen, einem Schema und einem schemaspezifischen Teil, getrennt durch einen Doppelpunkt, besteht. Beispielsweise lautet in *urn:nbn:ch:bel-9039* das Schema *urn* und der schemaspezifische Teil *nbn:ch:bel-9039*. Für das WWW sind vor allem die *Uniform Resource Locator* (URL) [18] relevant. Sie besitzen einen restriktiveren Aufbau als URIs und stellen eine Submenge dieser dar. Das Schema einer URL verweist auf den Mechanismus, der verwendet werden muss, um auf die Ressource zuzugreifen. Im WWW lautet das Schema daher meistens *http*, was auf das HTTP-Protokoll als Zugriffsmechanismus hindeutet.

2.1.2 HTTP

HTTP ist ein zustandsloses, auf dem Anfrage-Antwort-Prinzip basierendes Protokoll, mit welchem Repräsentationen von (HTTP-)Ressourcen übertragen bzw. manipuliert werden können. Eine HTTP-Nachricht besitzt einen sehr einfachen Aufbau. Bei einer HTTP-Anfrage (Listing 2.1) besteht sie aus der URI, die die Ressource identifiziert (Zeilen 1–2), der Methode, die auf die Ressource angewendet werden soll (Zeile 1), Metadaten, die weitere Bedingungen für die Anfrage spezifizieren (Zeilen 3–4), und einem optionalen Rumpf, der die eigentliche Nutzlast enthält (bei einer *GET*-Anfrage bleibt dieser leer). Die Metadaten werden als *Header* notiert.

Listing 2.1: Beispiel einer HTTP-Anfrage

```
1 GET /pp/zc.html HTTP/1.1
2 Host: localhost
3 User-Agent: SimpleHttpClient/1.0
4 Accept: text/xml,text/html;q=0.9,text/plain;q=0.8,*/*;q=0.5
```

HTTP-Anworten (Listing 2.2) bestehen aus einem Code, der dem Client Aufschluss über den Status der Anfrage gibt (Zeile 1), wiederum Metadaten (Zeilen 2–6) und einem optionalen Datenrumpf (Zeile 8).

Listing 2.2: Beispiel einer HTTP-Antwort

```
1 HTTP/1.1 200 OK
2 Date: Wed, 07 May 2008 07:30:58 GMT
3 Server: Jetty/4.2.x (Windows 2003/5.2 x86 java/1.6.0_02)
4 Content-Type: text/html
5 Transfer-Encoding: chunked
6 Content-Length: 19
7
8 <div>fragment</div>
```

Obwohl HTTP mehrere Methoden spezifiziert (*OPTIONS*, *GET*, *HEAD*, *POST*, *PUT*, *DELETE*, *TRACE* und *CONNECT*), werden im modernen WWW fast ausschließlich *GET* und *POST* verwendet. Mit *GET* kann eine Repräsentation von einer Ressource angefordert werden, mit *POST* können Daten zur Weiterverarbeitung an eine Ressource gesendet werden. Die Ressource fungiert in diesem Fall als Prozessor und wird verwendet, um beispielsweise neue Ressourcen zu erzeugen oder einen Dienst aufzurufen.

HTTP wurde so spezifiziert, dass es um neue Methoden und Header erweitert werden kann.

2.1.3 Datenformate im WWW

HTTP ist ein generisches Protokoll, es ist unabhängig von den bei den Repräsentationen verwendeten Datentypen. Primäres Datenformat ist HTML – eine textbasierte Auszeichnungssprache, die von einem Webbrowser gerendert werden kann. HTML-Dokumente können Links zu anderen Ressourcen enthalten. Dadurch entsteht ein riesiges Netz von verlinkten Ressourcen und schließlich das WWW als größtes Hypertext-Medium.

HTML ist für die visuelle Darstellung, jedoch nicht besonders für eine automatische Weiterverarbeitung durch Programme geeignet. Im Bereich der Maschine-zu-Maschine-Kommunikation haben sich daher vor allem die Protokolle der XML-Familie etabliert: Die *Extensible Markup Language* (XML) definiert eine durch *Tags* und *Attribute* erweiterbare Text-basierte Sprache, mit der sich Daten leicht strukturieren lassen. Mit *XML-Schema* lassen sich zudem Dialekte für bestimmte XML-Dokumenttypen definieren, und in der Folge sind die Dokumente validierbar.

Mit *Xlink* lassen sich Links zu XML-Dokumenten hinzufügen (analog zu HTML). Mit *XPath* können Teile eines XML-Dokumentes adressiert werden, und mit der *Extensible Stylesheet Language Transformation* (*XSLT*) können schließlich XML-Dokumente in andere Dokumenttypen transformiert werden. XML wird auf nahezu allen Plattformen und in fast allen Programmiersprachen unterstützt.

Die *JavaScript Object Notation* (JSON) ist ein weiteres Datenformat, welches in jüngster Zeit größere Bedeutung erlangt hat. Sie zeichnet sich durch ihre kaum zu unterbietende Einfachheit aus und unterstützt dabei die universellen Datentypen wie String, Integer, Float, Listen und Schlüssel-Wert-Paare. Sie wird vor allem im Browserumfeld eingesetzt, wo sie zusammen mit der *Ajax*-Technologie [20] zur clientseitigen, dynamischen Erzeugung von HTML eingesetzt wird. JSON wird von JavaScript, der Skriptsprache für Webbrowser, verstanden. Für den serverseitigen Einsatz gibt es (sehr leichtgewichtige) Implementierungen für fast alle Programmiersprachen [21].

2.2 Serviceorientierte Architekturen

2.2.1 Architektur und Paradigma

Serviceorientierte Architekturen (SOA) fanden in jüngster Zeit erhebliche Beachtung in den Bereichen der Geschäfts- und Internetanwendungen. Sie gelten als aussichtsreiches Paradigma, um bestehende und neue Anwendungen über administrative und technische Grenzen hinaus zu integrieren und zu nutzen und dabei gleichzeitig die Softwarekomplexität zu senken. Vor allem die Erfindung der *Web Services* verhalfen SOA zu hoher Popularität. Während Web Services jedoch konkrete Technologien liefern, beschäftigt sich SOA mit der dahinterliegenden Architektur, den Entwurfsrichtlinien und Prinzipien.

Es existieren unzählige Definitionen und Ansichten darüber, was zu einer grundlegenden SOA gehören muss und was nicht. Verantwortlich dafür ist der relativ späte Versuch, SOA als Architektur und Paradigma zu spezifizieren. Die Kernidee existiert schon wesentlich länger. Ein weiterer Grund ist darin zu finden, dass viele der SOA-Definitionen auf vorhandenen Technologien basieren, indem versucht wurde, von diesen zu generalisieren bzw. abzuleiten. Im Folgenden werden einige SOA-Definitionen bzw. Beschreibungen vorgestellt. Erl liefert in [22]:

> „Service-oriented architecture“ is a term that represents a model in which automation logic is decomposed into smaller, distinct units of logic. Collectively, these units comprise a larger piece of business automation logic. Individually, these units can be distributed.

Demnach bietet eine SOA ein Modell, in welchem verteilte logische Einheiten zusammen höher-

wertige Aufgaben erfüllen. Dodani erwähnt in [23] *standardisierte Schnittstellen* und *Nachrichten* (Dokumente) als wichtige Konzepte einer SOA:

> Service Oriented Architecture (SOA) enables flexible integration of applications and resources by: representing every application or resource as a service with a standardized interface, enabling the services to exchange structured information (messages, documents, 'business objects'), and coordinating and mediating between the services to ensure they can be invoked, used and changed effectively.

Dienste verstecken ihre Implementierung und können von *Geschäftsprozessen* verwendet werden:

> A service represents some functionality (application function, business transaction, system service, etc.) exposed as a component for a business process. The service itself is defined using a well-defined interface that exposes the functionality and hides the underlying implementation details.

Hashimi ergänzt in [24] die Konzepte eines Dienstes um *Plattformunabhängigkeit* und *dynamische Bindung*:

> A service in SOA is an exposed piece of functionality with three properties: The interface contract to the service is platform-independent, the service can be dynamically located and invoked, and the service is self-contained. That is, the service maintains its own state.

Dostal et al. führt in [25] die *sprachenunabhängige Nutzung* und *Wiederverwendbarkeit* als Kriterien auf:

> Unter einer SOA versteht man eine Systemarchitektur, die vielfältige, verschiedene und eventuell inkompatible Methoden oder Applikationen als wiederverwendbare und offen zugreifbare Dienste repräsentiert und dadurch eine plattform- und sprachenunabhängige Nutzung und Wiederverwendung ermöglicht.

Und schließlich stellen die Autoren in [26] eine ganze Reihe weiterer Anforderungen an Dienste und SOA:

> Services are discoverable and dynamically bound.
> Services are self-contained and modular.
> Services stress interoperability.
> Services are loosely coupled.

Services have a network-addressable interface.
Services have coarse-grained interfaces.
Services are location-transparent.
Services are composable.
Service-oriented architecture supports self-healing.

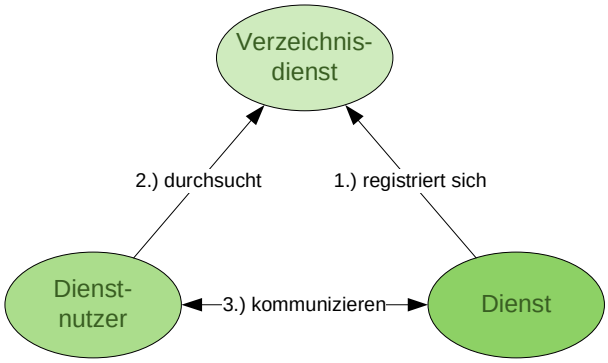

Abbildung 2.1: Komponenten einer SOA

Ohne sich auf eine dieser Definitionen festzulegen, wird für die weitere Arbeit SOA wie folgt charakterisiert: Serviceorientierte Architekturen sind verteilte Softwarearchitekturen, deren Komponenten *Dienste* und *Dienstnutzer* sind und deren Interaktionen auf Nachrichten basieren. Optional können sich Dienste bei einem Verzeichnisdienst registrieren, welches von Dienstnutzern durchsucht werden kann, wodurch Dienst und Dienstnutzer entkoppelt sind und erst zur Laufzeit dynamisch gebunden werden (*late binding*), siehe Abbildung 2.1. Dienste können einfache und komplexe Funktionalitäten, komplizierte Geschäftsprozesse oder ganze Anwendungen kapseln. Sie sind geeignet, alte Systeme in neue Systeme zu integrieren und das Gesamtsystem evolutionär zu entwickeln. Sie können sowohl auf einem Rechner, innerhalb eines Netzes oder auch über Netzwerk- und Unternehmensgrenzen hinaus miteinander agieren und neue, komplexere und höherwertige Anwendungen schaffen. Dienstnutzer verwenden Dienste entfernt, also außerhalb ihrer eigenen Laufzeitumgebung, indem sie Nachrichten an die Dienste schicken. Dienste besitzen Schnittstellen, welche in Dienstbeschreibungen spezifiziert werden. Eine solche Beschreibung enthält Angaben darüber, *was* der Dienst tut und welche Voraussetzungen dafür erfüllt sein müssen, nicht jedoch *wie* der Dienst die Schnittstelle implementiert.

Es gibt viele Möglichkeiten, eine SOA konkret umzusetzen, jedoch unterscheiden sich die Ansätze hinsichtlich ihrer Plattform- und Programmiersprachenabhängigkeit, der unterstützten Topologie, Skalierbarkeit, Robustheit, Sicherheit, Verifizierbarkeit usw. Es gibt bisher keine SOA-

Umsetzung, die sich domänenübergreifend durchgesetzt hat. Stattdessen adressieren einige Technologien wie *Home Audio Video Interoperability* (HAVi) oder *Universal Plug and Play* (UPnP) die Vernetzung und Gerätenutzung im Heimbereich, während andere wie Web Services vor allem für die internetweite Dienstnutzung zwischen Geschäftsanwendungen geeignet sind.

2.2.2 SOA als evolutionäres Produkt

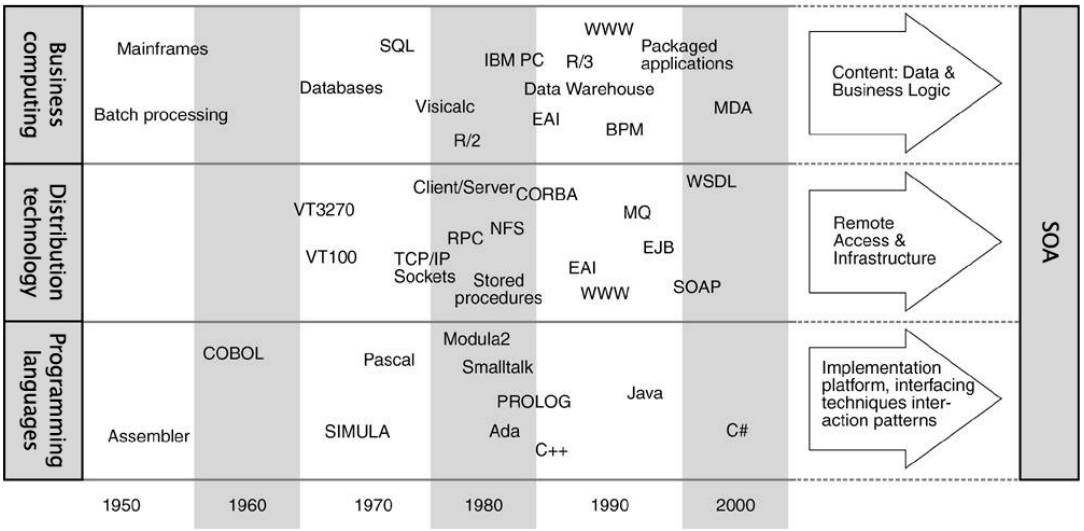

Abbildung 2.2: Serviceorientierung als Folge langfristiger Entwicklungen in den Bereichen Programmiersprachen, verteilte Systeme und Geschäftsinformatik [27]

Die Wurzeln von SOA liegen in der Entwicklung der Programmiersprachen, den verteilten Systemen und der Geschäftsinformatik [27]. Sie stellt daher die logische Fortsetzung dieser unterschiedlichen Entwicklungen dar und vereint deren Charakteristiken (Abbildung 2.2).

Abstraktion in Form von *funktionaler Abstraktion* und *Datenabstraktion* ist das wichtigste Konzept, welches SOA von den Programmiersprachen und der Softwareentwicklung erbt. Die *funktionale Abstraktion* wurde erstmals durch Fortran (1959) realisiert und stellt eine Leistung (einen Dienst) in Form einer abstrakten Funktion, Operation oder Prozedur zur Verfügung [28]. Die *Datenabstraktion* ist eine Abstraktion von einer Datenstruktur und deren Zugriffsoperationen.

Bei der *Objektorientierung* (OO) werden abstrakte Datentypen durch *Klassen* realisiert. Instanzen bzw. *Objekte* verwalten ihren eigenen Zustand und verstecken ihre Daten vor äußerem Zugriff (*Kapselung*). Weiterhin führt Objektorientierung die Konzepte *Vererbung* und *Poly-*

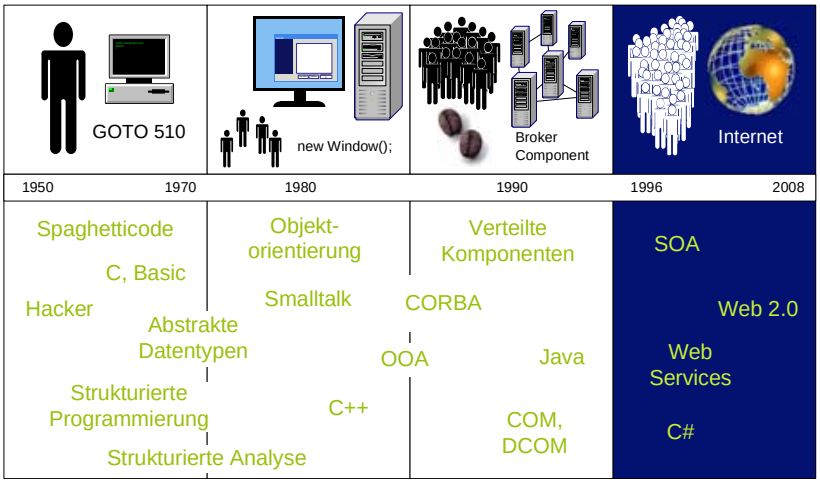

Abbildung 2.3: Evolution der Komplexität von Softwaresystemen, Programmiersprachen und Paradigmen (Ausschnitt)

morphie ein [29]. Vererbung ist eine spezielle Art von Relation zwischen Klassen, bei der eine Klasse die Datenstruktur und das Verhalten einer anderen Klasse erbt und diese um weitere Funktionalitäten ergänzen kann. Polymorphie erlaubt, dass die gleiche Nachricht durch unterschiedliche Klassen implementiert werden kann. Das Binden der Implementierung geschieht erst zur Laufzeit und ist von der verwendeten Schnittstelle (*Interface*) getrennt. Beide Konzepte ermöglichen unterschiedliche Arten der Wiederverwendung: Wiederverwendung von Verhalten und Daten (Vererbung) und Wiederverwendung von Schnittstellen (Polymorphie). Dienste einer SOA haben hier das Schnittstellenkonzept übernommen, da es Änderung des Verhaltens bei stabiler Schnittstelle und somit die stufenweise und unabhängige (Weiter-) Entwicklung von Diensten zulässt.

Komponenten sind Softwareelemente, die eine wesentlich gröbere Granularität besitzen als Objekte und explizit nur durch ihre Schnittstellen spezifiziert werden. Sie sind kontextfrei hinsichtlich ihrer Benutzung und können daher mit anderen Komponenten beliebig kombiniert werden. Sie teilen damit die Eigenschaft der Komponierbarkeit (*composability*) [30]. Dienste einer SOA sollen ebenfalls in einer Art entworfen werden, die die Integration in unterschiedlichen Kontexten erlaubt. Außerdem können Dienste durch Komposition zu neuen höherwertigen Diensten zusammengesetzt werden.

SOAs müssen über administrative Grenzen und Netzwerkgrenzen hinaus funktionieren. Dienste werden entfernt aufgerufen und genutzt. Das entfernte Aufrufen von Programmen bzw. Diensten sowie die Abstraktion der Systemumgebung findet seine Ursprünge im Fachgebiet der *Ver-*

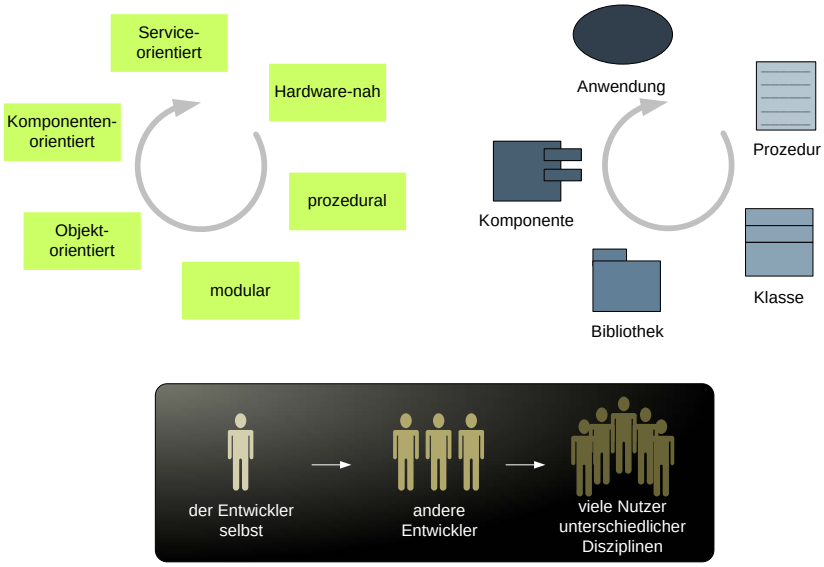

Abbildung 2.4: Stufen der Softwareentwicklung, der wiederverwendbaren Elemente und der Beteiligten, die von der Wiederverwendung profitieren

teilten Systeme. Ein verteiltes System ist ein aus Benutzersicht transparentes System, verbirgt also die Tatsache, dass es dabei aus einer Vielzahl von Computern besteht. Es gibt verschiedene Arten von Transparenzen wie Zugriffs-, Positions-, Migrations-, Relokations-, Replikations-, Nebenläufigkeits-, Fehler- und Persistenztransparenz [31].

Die ersten verteilten Systeme kommunizierten über Protokolle auf physikalischer Ebene. Sie wurden abgelöst durch spezielle Transportprotokolle wie TCP/IP, die u. a. die physikalische Sicht verbargen. *Remote Procedure Call* (RPC) war schließlich die erste Kommunikationsmiddleware, die tatsächlich von Medien, Netzwerkprotokollen und Plattformen abstrahieren konnte. Ein RPC ruft eine entfernte Prozedur oder Funktion in transparenter Weise auf und liefert deren Ergebnisse zurück.

Die *Common Object Request Broker Architecture* (CORBA) [32] erweitert RPC um Konzepte der Objektorientierung, indem entfernte Objekte erzeugt und aufgerufen werden können. CORBA spezifiziert zudem eine Reihe zusätzlicher Dienste (Naming, Transaktionen, Discovery usw.). Die feine Granularität der Schnittstellen (definiert mit der *Interface Definition Language* (IDL)) sowie die Komplexität der Technologie verhinderten jedoch eine universelle Verbreitung.

Obwohl RPC in seinen unterschiedlichen Ausprägungen (z. B. *Java Remote Method Invocation* (RMI) [33], *Enterprise Java Beans* (EJB) [34], Microsofts *Distributed Component Object Model* (DCOM) [35], CORBA usw.) die verteilte Umgebung für den Benutzer verbirgt, ist

das entsprechende Programmierparadigma, auf welchem es basiert (prozedural, objektorientiert usw.), weiterhin sichtbar. SOAs dagegen sollen nicht nur die o. g. Transparenzen unterstützen, sondern auch die jeweiligen Programmierparadigmen der Dienstimplementierungen verbergen.

Die Dienste einer serviceorientierten Architektur können Funktionen, Daten, Anwendungen und Prozesse kapseln. Dadurch sind sie hervorragend für den Bereich der Geschäftsinformatik geeignet und stellen dort Lösungskonzepte für eine Vielzahl von Problemen bereit:

- **Auslagerung** Das Auslagern von Tätigkeiten (*Outsourcing*), die nicht den Kernkompetenzen entsprechen, ist ein unumgängliches Mittel geworden, um schnell und kostengünstig auf sich ändernde Marktbedingungen und -anforderungen reagieren zu können. Dienste (Anwendungen und Prozesse) können extern durch Outsourcing erstellt und schließlich in die eigene SOA wieder zusammengeführt werden [36].
- **Geschäftsprozesse** Durch Anwendung des SOA-Konzeptes können Unternehmensprozesse in einzelne informationstechnische Geschäftsprozesse zerlegt und als autonome Dienste erstellt werden. In der Folge lassen sich Unternehmensprozesse dann durch Dienstkomposition abbilden und ausführen. Viel bedeutender ist jedoch die Möglichkeit, nachträglich auf neue Anforderungen reagieren zu können, die eine Änderung der Prozesse notwendig machen, indem nur die Komposition angepasst, nicht jedoch der einzelne Dienst geändert werden muss.
- **Vertikale Integration** Die vertikale Integration von der Auftragsabwicklung und Planung auf Geschäftsebene (*Enterprise Resource Planing* (ERP)) über die Kontrolle auf Produktionsebene (*Manufacturing Execution System* (MES)) und schließlich bis zur Abbildung auf Prozessebene (*Distributed Control System* (DCS)) kann in einer SOA einheitlich und in allen Schichten flexibel erfolgen. Dadurch wird man dem zunehmenden Trend weg von der Massenproduktion hin zur kundenindividuellen Massenfertigung gerecht [37].

Aufgrund der Nähe von Diensten zur Geschäftsinformatik gilt Serviceorientierung als potenziell erstes geeignetes Paradigma, welches von Partnern beider Seiten – sowohl technische als auch geschäftliche – verstanden werden kann [27]. Indem Geräte ebenfalls als Dienste einer SOA abgebildet werden, können sie ebenso einfach und mit den gleichen Mitteln in Geschäftsprozessen verwendet und Teil des Gesamtintegrationskonzepts werden.

2.3 Web Services

Ein *Web Service* ist prinzipiell jeder Dienst, der über das Web bzw. mit Web-Technologien nutzbar ist. Darunter fallen sowohl Techniken wie das Senden eines XML-Dokumentes via TCP als

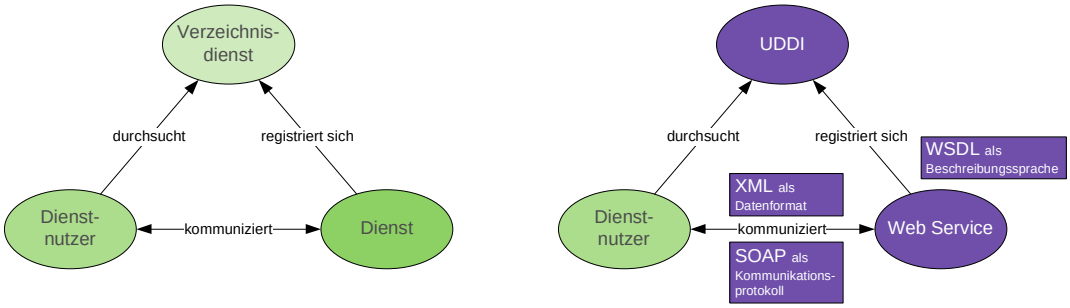

Abbildung 2.5: Web Services als Implementierung einer SOA

auch *Screen Scraping*[1]. Dennoch werden in der Regel mit Web Services die Dienste bezeichnet, die auf dem *Web Services Framework* aufbauen. Das Web-Services-Framework wird seit ca. 2000 entwickelt und bestand anfänglich im Wesentlichen aus den drei Protokollen SOAP, WSDL und UDDI, mit denen zusammen eine grundlegende serviceorientierte Architektur umgesetzt werden kann (Abbildung 2.5).

Durch die Anwendung in der Praxis stellten sich aber zunehmend Anforderungen heraus, die durch WSDL und SOAP alleine nur unzureichend abgedeckt werden konnten.

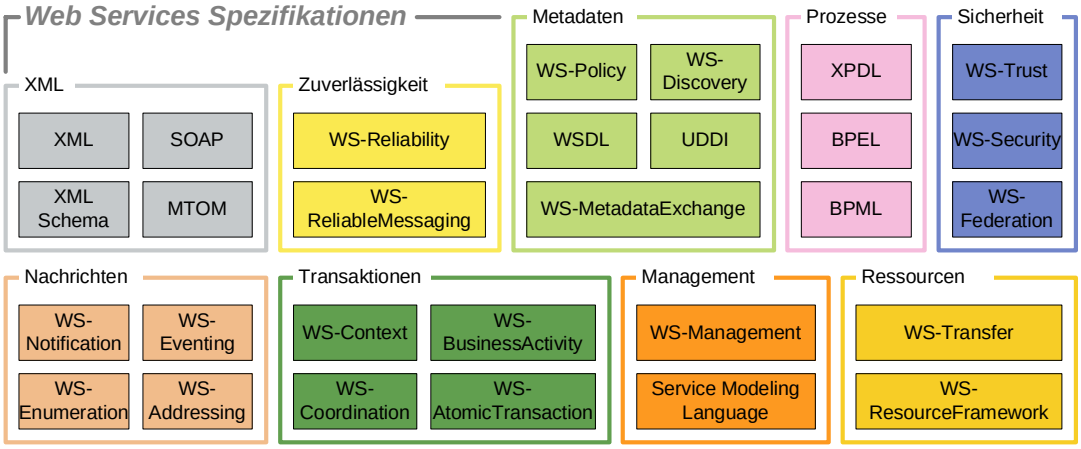

Abbildung 2.6: Web Services Spezifikationen (Auszug)

Daher folgten diesen auch als *Web Services 1. Generation* [22] bezeichneten Standards vor allem durch die Industrie getrieben eine Vielzahl neuer zum Teil überlappender und redundanter

[1] Technik, bei der Informationen aus unstrukturierten Webseiten extrahiert und zu strukturierten Informationen für die Weiterverarbeitung zusammengeführt werden

Web Services Spezifikationen. Abbildung 2.6 zeigt einen Auszug, eine detailliertere Übersicht ist in [38] zu finden. *Web Services 2. Generation* (auch: *WS-**) umfassen inzwischen mehr als 40 Protokolle für alle potenziellen Anforderungen verteilter Systeme wie Transaktionen, Zuverlässigkeit, Management und Workflow.

An der Spezifizierung sind große Softwarehersteller wie Microsoft, Intel oder BEA beteiligt, und sie geschieht weitestgehend offen und dezentral in unterschiedlichen Konsortien und Organisationen wie der *Organization for the Advancement of Structured Information Standards* (OASIS) [39], der *Web Services Interoperability Organization* (WS-I) [40], dem *World Wide Web Consortium* (W3C) [41] oder der *Internet Engineering Task Force* (IETF) [42].

Im Folgenden werden einige ausgewählte Protokolle vorgestellt, die für das weitere Verständnis dieser Arbeit notwendig sind.

2.3.1 Nachrichtenaustausch mit SOAP

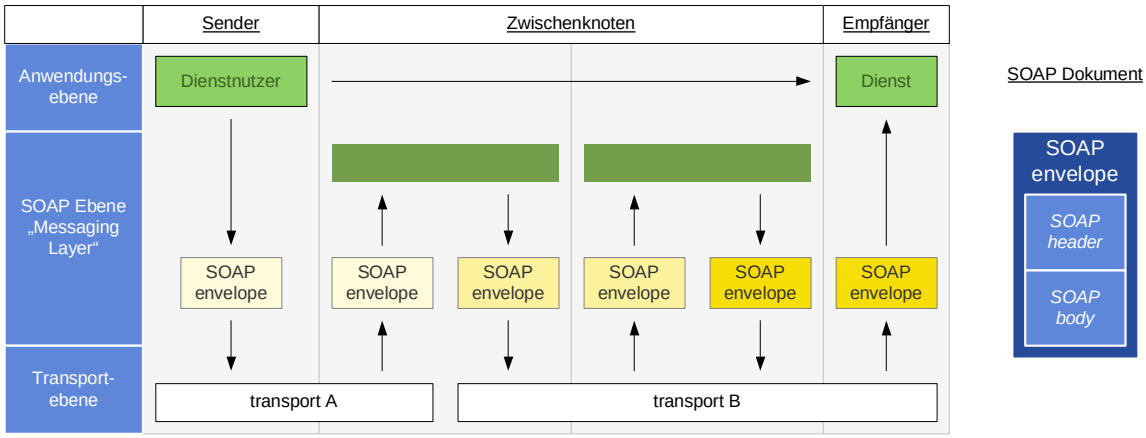

Abbildung 2.7: Struktur und Übertragung von SOAP-Dokumenten

Mit dem *Simple Object Access Protocol* (SOAP) [43] werden Nachrichten zwischen Dienst und Dienstnutzer unabhängig vom verwendeten Transportprotkoll übertragen. Damit übernimmt SOAP die Rolle des Anwendungsprotokolls im Web-Services-Framework. SOAP ist als XML-Grammatik spezifiziert und daher in erster Linie für die Übertragung von XML-Nachrichten geeignet (Listing 2.3). Eine SOAP-Nachricht (*SOAP Envelope*, Zeile 1) teilt sich auf in einen Kopf (*SOAP Header*, Zeilen 2–7) und einen Rumpf (*SOAP Body*, Zeilen 8–12) (Abbildung 2.7). Der Kopf enthält die Metadaten der Nachricht, beispielsweise Authentifizierungs- oder Transaktionsinformationen. SOAP stellt mit dem Kopf einen flexiblen Mechanismus zur Verfügung,

der es anderen Spezifikationen erlaubt, SOAP zu erweitern, indem sie zusätzliche Kopfelemente (*Header entries*, Zeilen 3–6) definieren können. Der Rumpf enthält die eigentliche Nachricht. Für den besonderen Fall, dass es sich dabei um eine Fehlernachricht handelt, hält SOAP vordefinierte Elemente bereit.

Listing 2.3: Beispiel einer SOAP-Nachricht [43]

```
1  <env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
2      <env:Header>
3          <n:alertcontrol xmlns:n="http://example.org/alertcontrol">
4              <n:priority>1</n:priority>
5              <n:expires>2001-06-22T14:00:00-05:00</n:expires>
6          </n:alertcontrol>
7      </env:Header>
8      <env:Body>
9          <m:alert xmlns:m="http://example.org/alert">
10             <m:msg>Pick up Mary at school at 2pm</m:msg>
11         </m:alert>
12     </env:Body>
13 </env:Envelope>
```

Die Aufteilung in Kopf und Rumpf verfolgt noch einen weiteren Zweck: Zwischen Sender und Empfänger können weitere Zwischenknoten (*SOAP intermediaries*) sitzen, die unterschiedliche Aufgaben wie Routen, Kompression, Cachen oder Loggen von SOAP-Nachrichten erfüllen. Für diese Zwischenknoten bietet der SOAP-Header eine einfache Möglichkeit zu erkennen, ob die Nachricht für sie bestimmt ist. Zwischenknoten können SOAP-Header entfernen, neue SOAP-Header oder sogar neue Zwischenknoten hinzufügen.

SOAP verfolgte, wie im Namen noch erkennbar, ursprünglich ein anderes Ziel: Es wurde als XML-RPC Standard entwickelt. SOAP unterstützt dazu die verschiedenen RPC-Konzepte wie Prozeduraufrufe und Ausnahmebehandlung auf XML-Basis. Obwohl es immer noch möglich ist, RPC mit SOAP zu realisieren, wird es im Fokus der Web Services nicht mehr verwendet.

SOAP 1.2 ist die aktuelle Version und stellt im Wesentlichen eine Verbesserung (Änderungen in Syntax und Klarstellungen der Semantik) der Vorgängerversion 1.1 dar.

SOAP wurde unabhängig vom verwendeten Transportprotokoll entworfen, was beispielsweise auch einen Einsatz über UDP, *Simple Mail Transfer Protocol* (SMTP) oder *Java Message Service* (JMS) erlaubt. Voraussetzung hierfür sind Spezifikationen, die definieren, wie SOAP auf das jeweilige Protokoll abgebildet werden soll. SOAP 1.1 und SOAP 1.2 enthalten jeweils ein Binding für HTTP, welches u. a. definiert, wie der Zweck der SOAP-Nachricht auf HTTP-Ebene zu kodieren ist. SOAP 1.1 verwendet hierzu den speziellen HTTP-Header *SOAPAction*, welcher eine URI enthält, die auf die Operation in der WSDL schließen lässt. SOAP 1.2 verwendet den *action*-Parameter des Medientypen *application/soap+xml*, der SOAP-Nachrichten identi-

fiziert [44]. HTTP ist (neben UDP) bislang das einzige Protokoll, für das SOAP ein Binding definiert[2].

Das *SOAP-over-UDP* Protokoll [45] trägt lediglich den Status eines Evaluierungsdrafts. Die verbindungslose Übertragung ist jedoch für einige Anwendungsfälle wie beispielsweise das Anmelden am Netzwerk, das Versenden von Ereignissen oder Video- und Audiodaten erwünscht oder sogar notwendig (z. B. Senden von Nachrichten an eine Multicast-Gruppe).

SOAP-over-UDP unterstützt das Versenden von SOAP 1.1- und SOAP 1.2-Nachrichten in UDP-Datagrammen sowohl als Unicast (in Anfragen und Antworten) als auch als Multicast (nur in Anfragen). Eine SOAP-over-UDP-URI ist am URI-Schema *soap.udp* erkennbar. Weiterhin definiert es einen *Retransmission Algorithmus*, der optional benutzt werden kann und der Unzuverlässigkeit beim Übertragen von Datagrammen begegnet. Dazu wird die gleiche Nachricht mehrfach versendet, wobei zwischen dem Senden zufällige Wartezeiten eingestreut werden, um Paketkollisionen zu vermeiden.

2.3.2 Beschreibung von Web Services mit WSDL

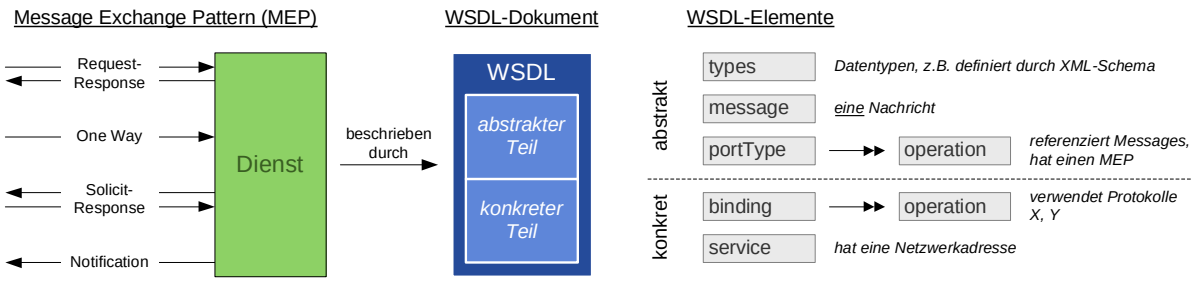

Abbildung 2.8: Struktur von WSDL-Dokumenten

Wichtige Charakteristik einer SOA ist die Dienstbeschreibung. Eine Dienstbeschreibung enthält alle Angaben, die ein Dienstnutzer benötigt, um den Dienst zu finden und mit ihm korrekt zu kommunizieren, und die ein Verzeichnisdienst (sofern verwendet) benötigt, um den Dienst bekannt zu machen. Die *Web Services Description Language* (WSDL) [46] übernimmt die Aufgabe der Dienstbeschreibung bei Web Services. Sie wurde 2001 von IBM und Microsoft zusammen initiiert und wird derzeit vom W3C weiterentwickelt.

[2] Abgesehen von einem vom W3C veröffentlichten Dokument, welches ein Binding für E-Mail spezifiziert. Dieses ist aber ausdrücklich als Demonstration gekennzeichnet, welches lediglich die Anwendung des Binding Frameworks von SOAP 1.2 illustriert und daher weder als offizielles Binding gewertet werden darf noch verwendet werden soll.

Listing 2.4: Beispiel einer WSDL-Service-Definition (Auszug) [46]

```
1  <?xml version="1.0"?>
2  <definitions name="StockQuote" xmlns="http://schemas.xmlsoap.org/wsdl/">
3
4      <types>...</types>
5
6      <message name="GetLastTradePriceInput">
7          <part name="body" element="xsd1:TradePriceRequest"/>
8      </message>
9
10     <message name="GetLastTradePriceOutput">
11         <part name="body" element="xsd1:TradePrice"/>
12     </message>
13
14     <portType name="StockQuotePortType">
15         <operation name="GetLastTradePrice">
16            <input message="tns:GetLastTradePriceInput"/>
17            <output message="tns:GetLastTradePriceOutput"/>
18         </operation>
19     </portType>
20
21     <binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
22         <soap:binding style="document"
23             transport="http://schemas.xmlsoap.org/soap/http"/>
24         <operation name="GetLastTradePrice">
25            <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
26            <input>
27                <soap:body use="literal"/>
28            </input>
29            <output>
30                <soap:body use="literal"/>
31            </output>
32         </operation>
33     </binding>
34
35     <service name="StockQuoteService">
36         <port name="StockQuotePort" binding="tns:StockQuoteBinding">
37            <soap:address location="http://example.com/stockquote"/>
38         </port>
39     </service>
40
41 </definitions>
```

Ein WSDL-Dokument ist ein XML-Dokument (Listing 2.4) und teilt sich auf in einen abstrakten und einen konkreten Teil (Abbildung 2.8). Der abstrakte Teil enthält Angaben zu den Operationen (Zeilen 15–18), *Port Types* (Sammlungen von Operationen, Zeilen 14–19), *Message Exchange Pattern* (MEP, Zeilen 16–17), Nachrichten (Zeilen 6–12) und Datentypen (Zeile 4), die der Dienst verwendet. Im konkreten Teil werden die Operationen und Daten auf konkrete

zu verwendende Protokolle wie SOAP und HTTP gebunden. Ein Binding (Zeilen 21–33) repräsentiert eine Möglichkeit bzw. Alternative, um mit dem Dienst kommunizieren zu können. Außerdem wird in Form einer Netzwerkadresse (Zeile 37) spezifiziert, wo der Dienst zu erreichen ist.

WSDL 1.1 unterscheidet zwischen vier verschiedenen *Message Exchange Pattern*, die den einer Operation zugrunde liegenden Nachrichtenfluss beschreiben: *One-Way*- und *Request-Response*-Operationen müssen von Dienstnutzern initiiert werden, während *Notification*- und *Solicit-Response*-Operationen vom Dienst selbst ausgehen.

WSDL ist ein offener Standard, der es erlaubt, zusätzliche Datenbeschreibungssprachen und Transportprotokolle mittels *Binding-Protokollen* zu integrieren. In der Praxis wird jedoch nahezu ausschließlich die Kombination SOAP/HTTP verwendet, was zum einen an der allgegenwärtigen Verfügbarkeit von HTTP, aber auch an fehlenden alternativen Protokollen liegt.

SOAP und WSDL sind von ihrem Entwurf her voneinander unabhängige Protokolle – SOAP ist ein Anwendungsprotokoll und mit WSDL kann die Schnittstelle eines Dienstes beschrieben werden. Um SOAP für WSDL und damit für einen Dienst als konkretes Protokoll verwendbar zu machen, gibt es das WSDL/SOAP Binding Protokoll als weitere Spezifikation, welches SOAP an WSDL bindet.

Es beschreibt u. a., wie eine Operation und ihre Parameter für das Senden einer SOAP-Nachricht kodiert und auf der anderen Seite wieder dekodiert werden müssen. Dafür gibt es fünf Alternativen: *RPC/encoded*, *RPC/literal*, *document/encoded*, *document/literal* und *Document/literal wrapped*. Jede dieser Alternativen hat Vor- und Nachteile, eine tiefere Diskussion ist unter [47] zu finden. Bei der RPC-Kodierung wird der Operationsname direkt als XML-Element notiert, wodurch sich die Nachricht auf Empfängerseite leichter zuordnen lässt als das bei der Document-Kodierung der Fall ist (reine Daten, kein Operationsname). Bei Anwendung der Literal-Kodierung kann die Nachricht von einem XML-Validator überprüft werden, bei der Encoded-Kodierung dagegen nur bei Teilen der Nachricht.

2.3.3 UDDI als öffentlicher Verzeichnisdienst

Um Web Services bekannt und potenziellen Nutzern zugänglich zu machen, müssen sie veröffentlicht werden. Diese Aufgabe übernimmt im Web-Services-Framework der ersten Generation die *Universal Description, Discovery and Integration* (UDDI) [48] Spezifikation. Sie gehört zu den älteren Protokollen, wurde seit 2000 innerhalb der OASIS Organisation entwickelt und später vor allem von Microsoft, IBM und SAP vorangetrieben und innerhalb der *UDDI Business Registry* (UBR) evaluiert. UBR ist eine Referenzimplementierung des UDDI-Standards, welche öffentlich zugängliche Dienste verwaltet. Die UBR wurde 2005 abgeschaltet, eine Alternative

bzw. die Fortsetzung von UDDI ist derzeit nicht geklärt.

Die *UDDI-Registry* übernimmt die Rolle einer Verzeichnisdienst-Komponente in einer SOA. In ihr können Dienstanbieter Metadaten ihrer Dienste ablegen. Dazu gehören Informationen wie über den Anbieter selbst (Firma, Adresse usw.), Dienstbeschreibungen (WSDL) und die Art und Weise (Sicherheit, Quality of Service, Transport), wie der Dienst zu erreichen ist. Daneben ist das Setzen von benutzerspezifischen Attributen möglich. Dienstnutzer können nach diesen Attributen und Metadaten suchen und erhalten UDDI-Einträge als Ergebnisse zurück. UDDI beinhaltet auch eine Benutzerverwaltung, so dass die Suchergebnisse in Abhängigkeit vom Suchenden beschränkt werden können.

UDDI ist selbst als Dienst realisiert und kann daher mit den gleichen Mechanismen genutzt werden (WSDL, SOAP) wie andere Web Services auch. Die Dienstschnittstelle einer UDDI-Registry umfasst u. a. Funktionen zum Anbieten, Suchen, Replizieren, Abonnieren von Diensten sowie für die Benutzerauthentifizierung.

2.3.4 Das Basic Profile

Die genannten, zu unterschiedlichen Zeitpunkten, entstandenen Spezifikationen lassen teilweise einen großen Interpretationsspielraum zu. Das führte in der Vergangenheit zu interoperablen Implementierungen [49] und somit am eigentlichen Ziel des Web-Services-Frameworks vorbei. Vor diesem Hintergrund bildete sich die *Web Services Interoperability Organization* (WS-I), deren Aufgabe darin besteht, Richtlinien für die korrekte Benutzung der Web Services Standards zu schaffen. Der WS-I Basic Profile Standard gilt als wichtigstes Dokument für die Schaffung interoperabler Web Services, da er die Zusammenarbeit zwischen den grundlegenden Protokollen SOAP, WSDL und UDDI spezifiziert.

2.3.5 Adressierung mit WS-Addressing

Der Nachrichtenaustausch bei Web Services der ersten Generation basiert in der Regel auf SOAP über HTTP. Das SOAP/HTTP-Binding nutzt dazu den Anfrage-Antwort-Mechanismus von HTTP, wodurch die Dienstaufrufe auf synchrone Kommunikation beschränkt sind. *WS-Addressing* [50] gilt daher als eines der grundlegendsten Protokolle der Web Services der zweiten Generation, da es komplexere Kommunikationsmuster für Web-Services-Interaktionen ermöglicht, die über den reinen synchronen Aufruf hinausgehen. Es setzt praktisch fast jedes der jüngeren Web-Services-Spezifikationen WS-Addressing voraus.

WS-Addressing definiert ein standardisiertes Modell, um Web Services und Nachrichten transportunabhängig zu addressieren. Dazu führt es die zwei neuen Konzepte *Endpoint Reference* (EPR) und *Message Information* Header (MI) ein. Mit EPRs können Endpunkte von Web

Services referenziert werden. Neben der obligatorischen Adresse können weitere Parameter, sogenannte *Reference Properties* und *Reference Parameter* angegeben werden, die den Endpunkt eindeutig identifizieren. Diese zusätzlichen Eigenschaften erlauben eine feinere Unterscheidung der möglichen Empfänger auf der Dienstseite, unabhängig von der verwendeten Adresse. Die Adresse kann sowohl eine HTTP-basierte URL, eine Mail-to-URI, eine logische URI oder ein anderer beliebiger *Internationalized Resource Identifier* (IRI) sein.

Listing 2.5: Beispiel einer SOAP-Nachricht mit Message-Information-Headers [50]

```
1  <S:Envelope xmlns:S="http://www.w3.org/2003/05/soap-envelope"
2      xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing">
3      <S:Header>
4          <wsa:MessageID>
5              uuid:6B29FC40-CA47-1067-B31D-00DD010662DA
6          </wsa:MessageID>
7          <wsa:ReplyTo>
8              <wsa:Address>http://business456.example/client1</wsa:Address>
9          </wsa:ReplyTo>
10         <wsa:To>http://fabrikam123.example/Purchasing</wsa:To>
11         <wsa:Action>http://fabrikam123.example/SubmitPO</wsa:Action>
12     </S:Header>
13     <S:Body>...</S:Body>
14 </S:Envelope>
```

MIs können in einer Nachricht verwendet werden, um die an der Nachricht beteiligten Endpunkte (Zeilen 7–10), den Zweck der Nachricht und die Nachricht selbst zu identifizieren (Listing 2.5). Nachrichten können mit einer *Message ID* (Zeilen 4–6) versehen und somit referenzierbar gemacht werden. Nachrichten-IDs sind der Schlüssel zu asynchronen Kommunikationsmustern, da Antworten nicht mehr sofort in einer HTTP-Antwort gesendet werden müssen, sondern auch später selbst über ein anderes Transportprotokoll wieder der ursprünglichen Anfrage zugeordnet werden können, wenn sie diese referenzieren. Weiterhin kann in der *Action*-Eigenschaft der Zweck der Nachricht gesetzt werden (Zeile 11). Dadurch muss dieser nicht mehr in der darunterliegenden Transportschicht kodiert werden, so wie es bisher bei HTTP mit dem *SOAPAction*-Header der Fall war.

Mit den MIs *From*, *To*, *ReplyTo* und *FaultTo* sind komplexere Interaktionsabläufe möglich, da zusätzlich neben Sender und Empfänger auch die Endpunkte (als EPR) angegeben werden können, an die die Antworten oder Fehler geschickt werden sollen (Abbildung 2.9).

2.3.6 WS-Discovery als dynamischer Verzeichnisdienst

Im Gegensatz zu UDDI als zentralem Verzeichnisdienst zielt *WS-Discovery* [51] auf dynamischere Umgebungen ab, in denen sich Dienste spontan (kurzfristig) veröffentlichen bzw. finden

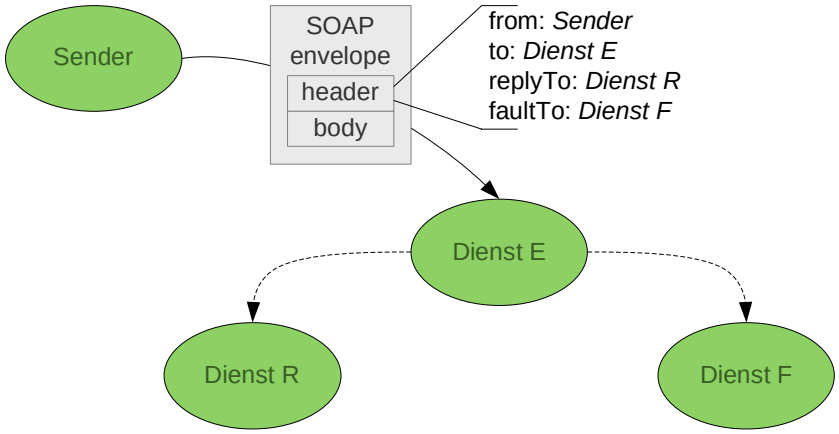

Abbildung 2.9: Komplexe Interaktionsmöglichkeiten für SOAP-Nachrichten durch die Verwendung von Message-Infomation-Header

lassen. Mit WS-Discovery ist es Dienstnutzern möglich, nach Diensten (*Target Services*) zu suchen, und umgekehrt können Dienste ihre Anwesenheit in einem Netzwerk bekanntgeben. Dazu definiert WS-Discovery eine IP-Multicastgruppe, an welche die Nachrichten geschickt werden. Netzwerkteilnehmer, die zu dieser Gruppe gehören, können die Discovery-Nachrichten empfangen. Dienste können außerdem *Typen* referenzieren, und Dienstnutzer können nach diesen suchen. Als Transportprotokoll kommt SOAP-over-UDP zum Einsatz. Einen möglichen Ablauf zeigt Abbildung 2.10.

Ein Dienst *TS1*, in diesem Fall eine Heizung/Klimanlage, tritt dem Netzwerk bei und sendet eine *Hello*-Nachricht an die Multicastgruppe. Diese enthält unter anderem die Typen *Heizung* und *Klima*, die der Dienst unterstützt. Die Nachricht wird hier nicht weiter ausgewertet, da es keinen Interessenten im Netzwerk gibt. Ein Dienstnutzer *DN* tritt ebenfalls dem Netzwerk bei und möchte die im lokalen Netz verfügbaren Heizungsgeräte ermitteln. Dazu schickt er eine *Probe*-Nachricht mit einem entsprechend spezifizierten Typ *Heizung* an die Multicastgruppe, welche sowohl von *TS1* als auch von *TS2* (einem weiteren Dienst) empfangen und ausgewertet wird. Während *TS2* die Anfrage nicht bedienen kann, sendet *TS1* eine *ProbeMatch*-Nachricht direkt an *DN* zurück, da er den angefragten Typ unterstützt. In der Nachricht hinterlegt *TS1* seine logische Adresse. Eine logische Adresse muss nicht physikalisch auflösbar sein und identifiziert einen Dienst daher unabhängig vom verwendeten Netzwerkprotokoll.

Um die logische Adresse schließlich in eine Transportadresse aufzulösen, sendet *DN* eine *Resolve*-Nachricht direkt an *TS1*, welcher mit einem *ResolveMatch* antwortet.

WS-Discovery spezifiziert, wonach gesucht werden kann und wie diese Suche auf Dienstsei-

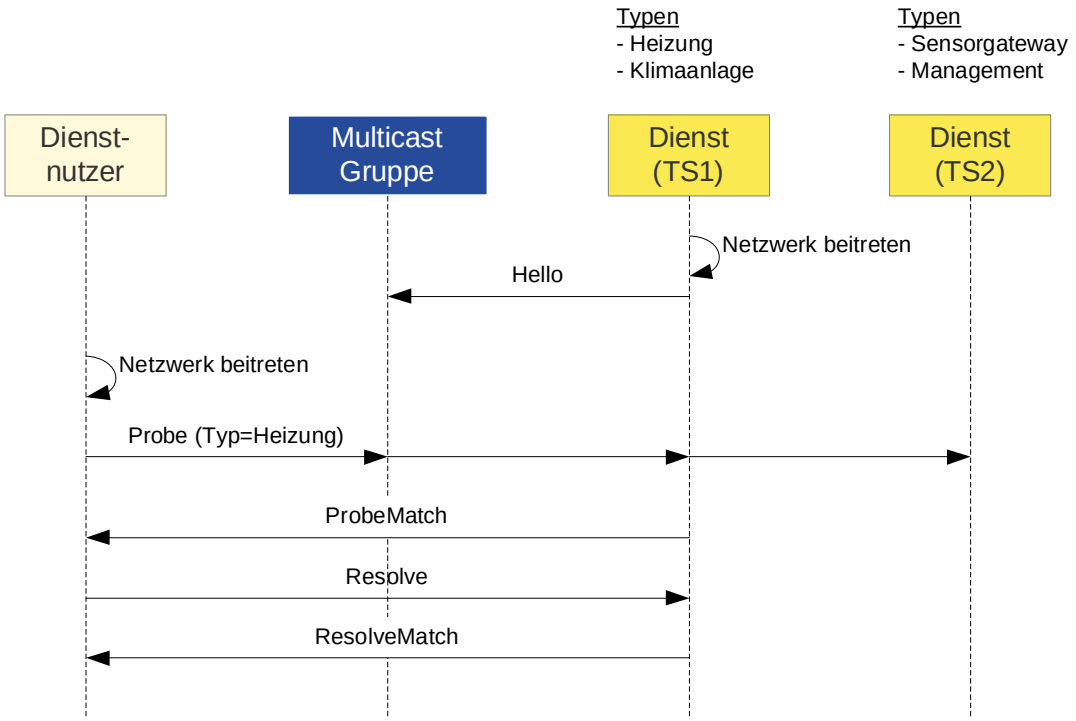

Abbildung 2.10: WS-Discovery Nachrichten in einem Netzwerk

te abgeglichen (*Matching*) werden kann. Es sind Suchen nach Diensttypen (*Types*) und nach Gültigkeitsbereichen (*Scopes*) möglich.

Das dargestellte Szenario demonstriert das dynamische Entdecken von Diensten und ist für lokale Netzwerke mit wenigen Beteiligten sehr gut geeignet. Um WS-Discovery jedoch auch in Netzwerken mit einer sehr hohen Zahl an Diensten oder über lokale Netzwerkgrenzen hinaus einsetzen zu können, definiert die Spezifikation einen *Suppression*-Mechanismus und mit dem *Discovery Proxy* einen weiteren Endpunkt (bzw. Diensttyp). Der Discovery Proxy ist ein optionaler Stellvertreter für Dienste, übernimmt deren Discovery-Funktionen und ist im Netz selbst als Dienst abgebildet. Sucht ein Dienstnutzer nach Diensten an der Multicastgruppe, sendet ein Discovery Proxy eine Suppression-Nachricht an den Dienstnutzer, woraufhin dieser zukünftige Anfragen nur noch direkt an den Discovery Proxy stellen darf.

Da WS-Discovery auf SOAP-over-UDP aufsetzt und UDP ein verbindungsloses Transportprotokoll ist, musste auf Anwendungsebene ein zusätzlicher Mechanismus geschaffen werden, mit dem die ursprüngliche Reihenfolge der UDP-Datagramme wieder hergestellt werden kann. WS-Discovery definiert dazu einen *Application Sequence* SOAP-Header, mit dem Nachrichten

fortlaufend durchnummeriert werden können.

2.3.7 Ereignisse mit WS-Eventing

Bei Web Services der ersten Generation mussten Ereignisse zwischen Web Services über einen individuellen Mechanismus abonniert und veröffentlicht werden. Dieses führte zu proprietären Lösungen. Ein Dienstnutzer, der Dienste von verschiedenen Partnern konsumieren wollte, war deshalb darauf angewiesen, sämtliche unterschiedlichen Mechanismen zu unterstützen. *WS-Eventing* [52] ermöglicht eine standardisierte, ereignisbasierte Middleware für Web Services.

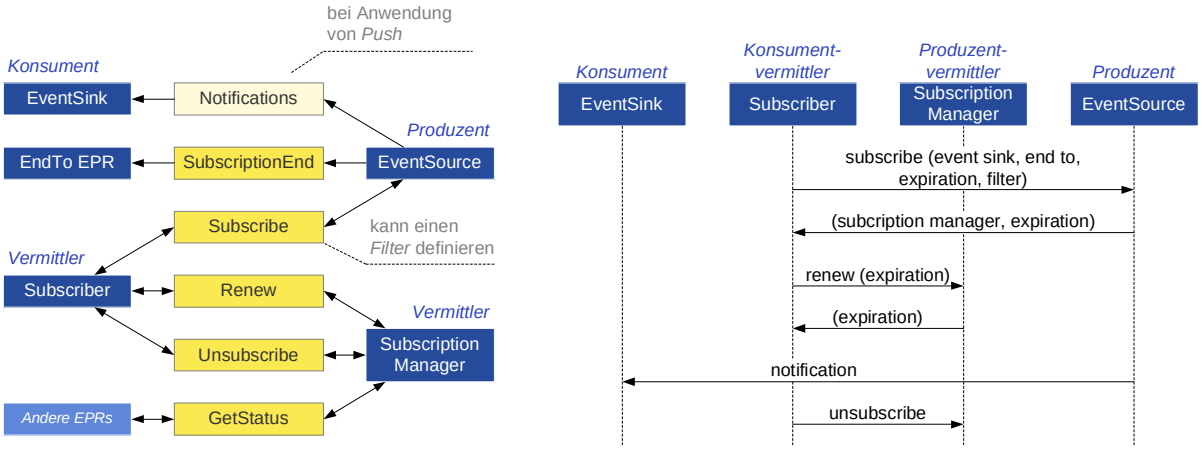

Abbildung 2.11: Nachrichten, Endpunkte und typischer Ablauf bei der Verwendung von WS-Eventing

Interessenten (*Subscriber*) abonnieren Ereignisse beim Produzenten (*Event Source*) (Abbildung 2.11). Dabei können sie einen *Filter* angeben, der den Umfang der Ereignisnachrichten beschränkt. Der Produzent darf in der Folge eine Ereignisnachricht nur dann verschicken, wenn der auf die Nachricht angewendete Filter *true* ergibt. WS-Eventing definiert einen auf XPath basierenden Filter, der einen Prädikatausdruck enthält. Anderen Spezifikationen ist es jedoch ausdrücklich erlaubt, auch andere Filter zu definieren.

Weiterhin enthält die Nachricht eine Angabe darüber, wie die Ereignisse veröffentlicht werden (*Delivery Mode*), sowie eine Anforderung für die Gültigkeitsdauer des Abonnements. Diese kann vom Dienst akzeptiert oder geändert werden und spezifiziert entweder eine befristete oder eine unbefristete Dauer (*Lease-Konzept*). Der Veröffentlichungsmechanismus ist erweiterbar gehalten. WS-Eventing spezifiziert als Voreinstellung den *Push*-Mechanismus, bei welchem die Nachrichten direkt an die Konsumenten gesendet werden. Wird Push verwendet, muss daher

zusätzlich noch der Endpunkt des Konsumenten angegeben werden (*Notify To*).

Die Antwort referenziert einen *Subscription Manager*, über welchen alle weiteren Interaktionen wie Verlängerung oder Beendigung des Abonnements laufen. Da der Subscription-Manager in der Regel *ein* Dienst ist, jedoch viele Abonnements verwalten muss, kann der Subscription-Manager-Endpunkt mit einem Reference-Property versehen werden, der eine das Abonnement identifizierende Nummer enthält. WS-Eventing spezifiziert für diesen Fall das optionale *Identifier*-Element.

WS-Eventing definiert das Attribut *EventSource="true"*. Mit diesem können Port-Types in einem WSDL-Dokument markiert werden. Dadurch wird der Dienst, der diesen Port-Type implementiert, als eine Event-Source ausgezeichnet, das heisst, er kann Subscription-Nachrichten empfangen. Wenn ein Subscriber eine Event-Source abonniert, so gilt das Abonnement für *alle* Notification- und Solicit-Response-Operationen in den mit *EventSource* markierten Port-Types.

2.4 Representational State Transfer

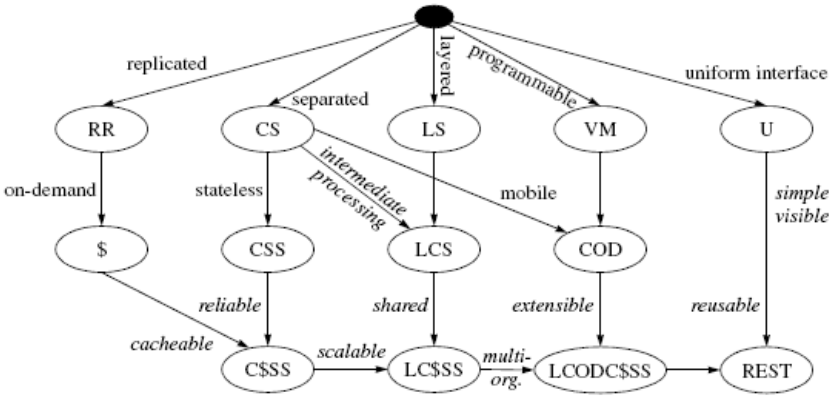

Abbildung 2.12: Ableitung des REST-Architekturstils [53]

Der Begriff *Representational State Transfer* (REST) wurde erstmals von Fielding in [53] geprägt und ist der Versuch, die dem *World Wide Web* (WWW) zugrundeliegende Architektur zu beschreiben. REST ist eine Kombination mehrerer *Architekturstile*.

Ein Architekturstil definiert Bedingungen für Architekturelemente (Komponenten, Konnektoren, Daten). Beispielsweise verwendet der bekannte *Uniform Pipe and Filter* Stil die Komponente *Filter* sowie den Konnektor *Pipe* und definiert für Filter die Bedingung, völlig unabhängig von anderen Filtern zu sein. Aufgrund dieser Bedingung ergibt sich die Eigenschaft, Filter und

Pipes in *beliebiger* Kombination miteinander verketten zu können.

Aus den Bedingungen, die von Architekturstilen definiert werden, resultieren Eigenschaften, die die Zielarchitektur aufweist. Vor diesem Hintergrund muss REST betrachtet werden, da es mehrere Stile vereint und somit seinen Architekturelementen zahlreiche Bedingungen auferlegt. Die entsprechende Ableitung des REST-Stils zeigt Abbildung 2.12, wobei in den Ellipsen die Stile und an den Pfeilen die daraus resultierenden Eigenschaften notiert sind. Für eine detaillierte Erläuterung sei auf [53] verwiesen.

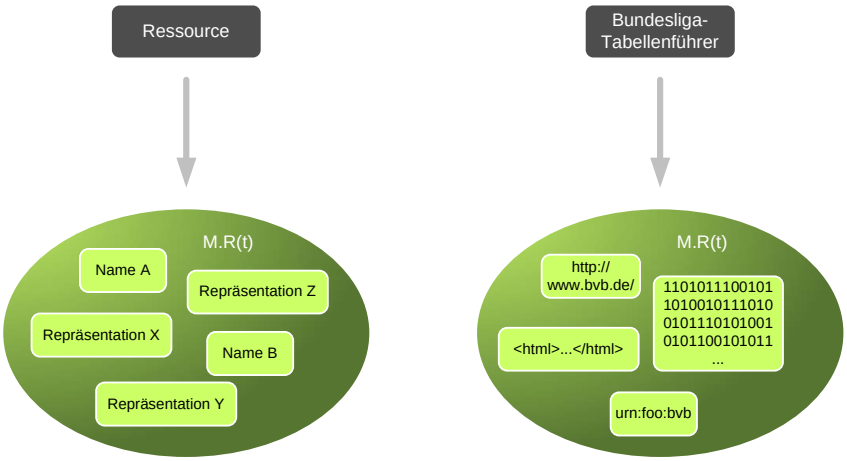

Abbildung 2.13: Abbildung einer Ressource auf eine Menge von Namen und Repräsentationen

Grundlegende Datenelemente in REST sind *Ressourcen* und *Repräsentationen*. Eine Ressource kann jedes Konzept oder jede Information sein, deren Semantik sich nicht ändert. Eine Repräsentation ist eine selbstbeschreibende Folge von Bytes, die den aktuellen oder beabsichtigten Zustand der Ressource darstellt. Mathematisch betrachtet bildet eine Ressource R zu einem bestimmten Zeitpunkt t auf eine Menge $M_{R(t)}$ von gleichwertigen Elementen (Ressourcennamen und/oder Repräsentationen) ab (Abbildung 2.13). Während die Elemente und damit die Menge $M_{R(t)}$ sich temporal ändern können, bleibt die Semantik der Ressource über die Zeit konstant.

Beispielsweise steht das Konzept „Bundesliga-Tabellenführer“ für den aktuellen Tabellenführer der Bundesliga, welcher in einer Woche „Bayern München“ und in einer anderen Woche „Borussia Dortmund“ sein könnte. Repräsentationen sind notwendig, um Ressourcen für andere Komponenten interpretierbar (darstellbar, verarbeitbar usw.) zu machen. Die Art der Repräsentation wird von REST nicht beschränkt. Im vorliegenden Beispiel könnten die Vereine ebenso durch HTML (z. B. die Vereinsseite) als auch durch ein Mannschaftsbild repräsentiert werden.

Neben den Repräsentationen können auch Namen wie ein *Uniform Resource Locator* (URL) in *http://www.bvb.de/* oder ein *Uniform Resource Name* (URN) in *urn:foo:bvb* verwendet werden und somit Elemente der Menge sein, auf welche die Ressource zu einem bestimmten Zeitpunkt abbildet. Die Elemente einer Ressource können sich also ändern, während die Semantik (hier: der aktuelle Tabellenführer der Bundesliga) der Ressource unverändert bleibt.

Der Datentransfer innerhalb einer REST-Architektur geschieht mittels *Client-* und *Server-Konnektoren*, die die verschiedenen Architektur-Komponenten (*User Agent*, *Origin Server*, *Proxy*, *Gateway*) miteinander verbinden. Clients initiieren die Kommunikation, indem sie eine Anfrage an einen Server schicken. Server warten auf Anfragen von Clients, nehmen sie entgegen und beantworten sie. Kennzeichnend für REST ist, dass die Server zustandslos sind und keine Kenntnis über die Clients und ihre Anfragen über die Dauer eines Anfrage-Antwort-Zyklus hinaus haben (*Client-Stateless-Server*-Stil). Ein solcher Kommunikationszyklus ist in sich abgeschlossen und unabhängig von früheren Anfragen. Daher muss eine Anfrage immer alle Daten enthalten, die der Server benötigt, um auf die Anfrage antworten zu können. Das Prinzip der zustandslosen Server erhöht die Skalierbarkeit des Systems. Auf der anderen Seite erhöht dieser Ansatz die Menge der Daten, die vom Client zum Server übertragen werden müssen, da der Server auf keinen vorhandenen Kontext bezüglich des Clients zurückgreifen kann.

Die zwei wichtigsten Architekturkomponenten in REST sind *User Agent* und *Origin Server*. Sie sind jeweils die ultimativen Empfänger von Anfragen und Antworten, müssen jedoch nicht direkt miteinander über Konnektoren verbunden sein. Statt dessen können je nach Topologie beliebige Zwischenkomponenten (*intermediaries*) in Form von *Proxies* und *Gateways* zwischengeschaltet sein. Proxies und Gateways kapseln weitere Dienste (z. B. DNS, Datentransformation, Sicherheit usw.). Die Verwendung von Zwischenkomponenten erfolgt für User-Agents und Origin-Servers transparent.

Um die Performance für User-Agents zu erhöhen und gleichzeitig den Datentransfer im Netzwerk zu verringern, können Antworten durch Komponenten gepuffert und für spätere Anfragen wiederverwendet werden (*Cache*-Stil). Dazu können in REST Cache-Konnektoren sowohl an Client- als auch Server-Konnektoren angedockt werden, die die Antworten gegebenenfalls speichern. Wird zu einem späteren Zeitpunkt von einem Client eine identische Anfrage geschickt, kann der Cache-Konnektor die (sofern noch) gültige Antwort direkt zurückschicken und somit auf ein Weiterleiten der Anfrage verzichten. Das Cachen wirkt sich dadurch positiv auf den Datentransfer aus und kann ihn u. U. sogar verhindern, wenn sich der Cache-Konnektor auf der gleichen Maschine befindet wie der User-Agent.

Der bedeutendste Architekturstil von REST liegt in der Forderung nach einer einheitlichen Schnittstelle zwischen *allen* beteiligten Architekturkomponenten (*Uniform-Interface*-Stil). Da-

durch wird das Gesamtsystem vereinfacht (homogenisiert) und die Sichtbarkeit der Interaktionen verbessert. Sichtbarkeit bezieht sich hier auf die Interpretierbarkeit der Interaktionen von außen – bedingt durch die einheitliche Schnittstelle (z. B. Überwachung durch Firewalls). Die uniforme, von REST definierte, Schnittstelle ist durch vier Bedingungen gekennzeichnet:

- **Identifizierung von Ressourcen** Ressourcen, die in den Interaktionen zwischen den Komponenten einbezogen sind, werden durch Ressourcenbezeichner, nicht durch eine Repräsentation, identifiziert ($M_{R(t)}$ muss jeweils einen solchen enthalten).
- **Manipulation von Ressourcen durch Repräsentationen** Um den Zustand (den Wert) einer Ressource zu lesen, zu ändern oder zu überschreiben, werden Repräsentationen der Ressource zwischen den Komponenten übertragen.
- **Selbstbeschreibende Nachrichten** Alle Anfragen an Komponenten enthalten den die Ressource identifizierenden Ressourcenbezeichner, für die die Anfrage gilt, weiterhin Kontrolldaten, die den Zweck der Nachricht genauer beschreiben (u. a. die Methode, die die Aktion beschreibt, die auf der Ressource ausgeführt werden soll), und optional (je nach Methode) eine Repräsentation. Antworten von Komponenten enthalten wiederum Kontrolldaten, Metadaten, die die Antwort genauer beschreiben (z. B. den Datentyp der Repräsentation), und optional eine Repräsentation. Die Repräsentationen, die zwischen den Komponenten transferiert werden, enthalten entweder den aktuellen oder den erwünschten Zustand der Ziel-Ressource oder aber die Repräsentation einer anderen Ressource, der für die Verarbeitung an eine Ressource gesendet wird. *Selbstbeschreibend* bezieht sich auf die Bedingung, dass in REST alle Interaktionen unabhängige in sich abgeschlossene Anfragen und Antworten sind.
- **Hypermedia als Zustandsmaschine der Anwendungen** Als *Hypermedia* werden Dokumente bezeichnet, die Links auf andere Dokumente enthalten. Hypermedia-Dokumente können neben Hypertext-Dokumenten, die den Dokumententyp auf Text beschränken, beispielsweise XML, HTML, Flash-Animationen oder Powerpoint-Präsentationen sein. In REST wird Hypermedia als Zustandsmaschine verwendet, da die Dokumente (die Repräsentationen) jeweils Links zu den nächsten Ressourcen, die die nächsten möglichen Zustände darstellen, enthalten können. Ein User-Agent kann von einem Origin-Server nach jeder Interaktion in einen neuen Zustand versetzt werden, wodurch sich Techniken wie *Load Balancing* leicht verwirklichen lassen. Am besten lassen sich die verlinkten Dokumente als Graph veranschaulichen, in welchem jeder Knoten die möglichen Alternativen zu den nächsten Knoten als Links präsentiert. Die Sicht des User-Agents ist in der Regel jedoch immer nur auf einen Knoten beschränkt.

2.5 Zeroconf

Mit *Zeroconf* [54] wird eine spezielle Technologie bezeichnet, die eine wichtige Lücke im Umgang mit dynamisch vernetzten und dezentral organisierten Geräten in lokalen IP-Netzwerken schließt. Die Anwendung von Zeroconf erlaubt Geräten oder Rechnern die *konfigurationsfreie* Zuweisung einer eindeutigen IP-Adresse, eines eindeutigen Hostnamens sowie das Durchsuchen und Überwachen des Netzwerkes nach und von Diensten. Die Besonderheit dieser Technologie liegt in der ausschließlichen Nutzung etablierter Internet- bzw. Netzwerkstandards, die lediglich minimal unter Beibehaltung der Kompatibilität angepasst wurden.

Die automatische Zuweisung einer IP-Adresse greift, wenn kein *Dynamic Host Configuration Protocol* (DHCP) Server [55] zur Verfügung steht. In diesem Fall konfiguriert Zeroconf eine zufällige IP-Adresse aus dem *link-local* Bereich (für IPv4 169.254.[1-254].[0-255]) und überprüft mittels eines Algorithmus, welcher auf dem *Address Resolution Protocol* (ARP) basiert, ob die Adresse im lokalen Netzwerk eindeutig ist [56]. Im Konfliktfall wird diese Vorgehensweise wiederholt.

Das *Domain Name System* (DNS) bildet die Grundlage für Zeroconf. DNS ist eine hierarchisch aufgebaute Datenbank, die internetweit funktioniert und ihre Datensätze (*Records*) auf *DNS-Servern* speichert. Ein solcher Datensatz besteht immer aus einem *Namen*, einem *Typ* (*Record Type*) und einem *Wert*, der dem Namen zugewiesen ist. Beispielweise hat sich im Internet die Adressierung mit eindeutigen Hostnamen durchgesetzt, die erst bei Bedarf in IP-Adressen aufgelöst werden. IP-Adressen (Record-Type *A*) lassen sich mit DNS auf Domainnamen abbilden: *www.apple.com A 17.254.0.91*.

Da in lokalen Netzwerken kein Internetzugang und kein vorkonfigurierter Unicast-basierter DNS-Server vorausgesetzt werden kann, spezifiziert Zeroconf mit *Multicast DNS* (mDNS) [57] einen auf Multicast basierenden Mechanismus (IP 224.0.0.251, UDP-Port 5353) unter Beibehaltung der in DNS definierten Semantiken. *DNS-Clients* verwenden die Multicastgruppe, um DNS-Anfragen durchzuführen, gleichzeitig nutzen sie aber auch die DNS-Antworten anderer DNS-Clients, als wären es eigene Anfragen gewesen. Dieser und andere Mechanismen, die zur Reduzierung der Netzwerklast beitragen, sind ebenfalls Bestandteil von mDNS.

Um sich einen eigenen Hostnamen zuzuweisen, steht für lokale Netzwerke die spezielle *Top-Level-Domain* (TLD) *.local.* zur Verfügung, die für Hostnamen im link-local Bereich verwendet werden kann. Die Eindeutigkeit des Namens wird mit mDNS überprüft, indem eine DNS-Anfrage gesendet wird, die den gewählten Namen an die gewählte IP-Adresse bindet. Im Konfliktfall muss ein anderer Name gewählt werden.

Da DNS lediglich Namen und Typen auf Werte abbildet, liegt es nahe, DNS auch für die

Registrierung von Diensten zu verwenden. Mit *SRV* existiert ein solcher Record-Type bereits. Dieser bildet Dienstnamen (inkl. Domainname) jeweils auf einen Hostnamen und einen Port ab, z. B.: *_http._tcp.local. SRV jd.local., 80.* Eine der Anforderungen von Zeroconf war es jedoch, Dienstinstanzen von Diensttypen auflisten zu können, um Benutzern diese repräsentieren zu können. Bei SRV wären das aber Hostnamen und Ports. Daher spezifiziert Zeroconf mit dem *DNS Service Discovery* (DNS-SD) Protokoll, dass Dienstinstanzen über eine *Reverse-Anfrage* (Record Type *PTR* für Pointer) und der Host bzw. Port der Dienstinstanzen über eine SRV-Anfrage gefunden werden. Reverse-Anfragen ermitteln im Gegensatz zu allen anderen DNS-Anfragen eine *Liste* von Namen zu einem gegebenen Wert. Mit DNS-SD und mDNS lassen sich Dienste finden, und es lässt sich das Netzwerk auf neue, aktualisierte und entfernte Dienste überwachen.

2.6 Web Application Description Language

Die *Web Application Description Language* (WADL) [58] ist eine formelle, in XML notierte Beschreibungssprache, die für die Spezifizierung von HTTP-basierten Diensten herangezogen werden kann. Mit WADL können die erlaubten HTTP-Methoden für bestimmte Ressourcen oder Gruppen von Ressourcen festgelegt werden. Darüber hinaus können die MIME-Typen der Repräsentationen eingegrenzt, bei Verwendung von XML können XML-Schemas angegeben und für Anfrageparameter, die an URLs angehängt werden, können erlaubte Datentypen oder Wertemengen angegeben werden.

Insgesamt eignet sich WADL zur Erzeugung von client- und serverseitigem Code, der einen Teil der Validierung zur Laufzeit abnimmt oder Methodenrümpfe zur Verfügung stellen kann. Ein Codegenerator für Clientanwendungen in Java ist unter [59] zu finden.

2.7 Universal Plug and Play

Das *Universal Plug and Play* (UPnP) Forum [60] wurde 1999 von Microsoft zusammen mit einigen anderen Firmen ins Leben gerufen. Ziel war es, einen Standard für die Integration von Geräten in kleine lokale Netzwerke zu schaffen, der weitestgehend unabhängig von Programmiersprachen und Technologien ist, die juristisch einer bestimmten Firma gehören oder deren Nutzung anderweitig eingeschränkt ist. Die Spezifikation ist in der *UPnP Device Architecture* (UDA) [61] festgehalten.

UPnP basiert auf dem Client-Server-Konzept, wobei *UPnP Devices* (UPnP-Geräte) ihre *UPnP Services* (Dienste) in einem Netzwerk anbieten. *UPnP Control Points* können nach Geräten/Diensten suchen, sie benutzen und Ereignisse abonnieren. Der UPnP-Protokollstapel (Abbildung 2.14) besteht aus drei aufeinander aufbauenden Schichten: Die untere ist die UDA.

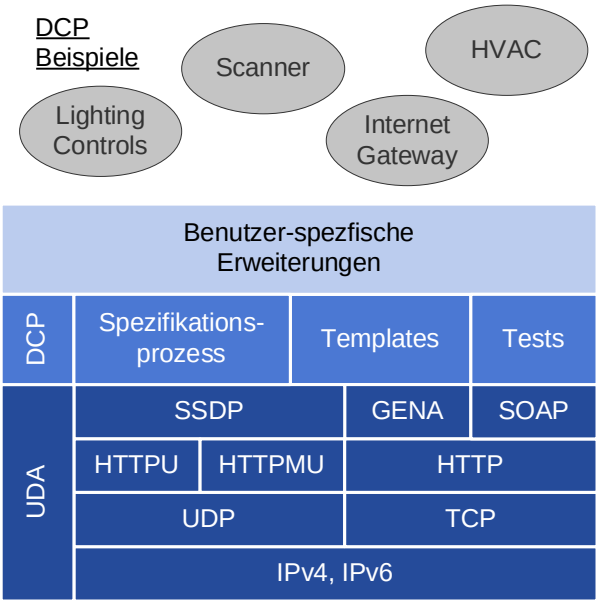

Abbildung 2.14: Der UPnP Protokollstapel

Sie spezifiziert die Addressierung, Entdeckung, Steuerung, Beschreibung, Ereignismechanismen und Präsentation von UPnP-Geräten, Diensten und Control-Points. Die *Device Control Protocol* (DCP) Schicht besteht aus einer losen Sammlung von standardisierten Geräte- und Diensttypen. Die DCPs durchlaufen einen festgelegten Standardisierungsprozess innerhalb des UPnP-Forums, bevor sie offiziell den Status eines Standards erhalten. Während dieser Phase kann jeder die Spezifikationen sichten und Einfluß auf ihre weitere Entwicklung nehmen. Die dritte Schicht ist dem indivuellen Anwender vorbehalten. In diese können zusätzliche Dienste aufgenommen werden. Ermöglicht wird diese Schicht durch einen in die Geräte- und Dienstbeschreibungen eingebauten Erweiterungsmechanismus.

Damit ein Gerät in einem UPnP-Netzwerk benutzt werden kann, muss es eine eindeutige IP-Adresse besitzen. Für den Fall, dass kein DHCP-Server zur Verfügung steht, verwendet UPnP für die Zuweisung einer eindeutigen IP-Adresse während der *Addressing*-Phase den gleichen Algorithmus wie Zeroconf [56].

Nachdem das Gerät eine IP-Adresse besitzt, kann es seine Anwesenheit mittels *Discovery* bzw. *Advertisement* bekanntgeben (Abbildung 2.15). UPnP verwendet für diesen Zweck das *Simple Service Discovery Protocol* (SSDP) [62], welches HTTP um einige HTTP-Header erweitert und UDP als Transportprotokoll benutzt. Die *Advertisements* haben eine begrenzte Gültigkeitsdauer, weshalb sie in regelmäßigen Abständen erneuert werden müssen (*Lease-Konzept*). Control-

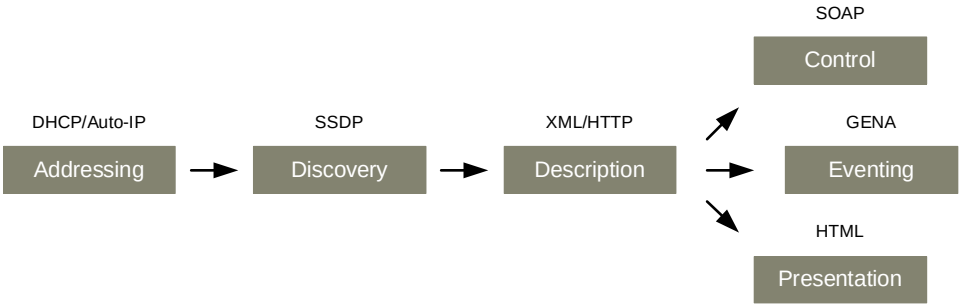

Abbildung 2.15: Sequenzieller Ablauf der UPnP-Phasen

Points können mit dem gleichen Protokoll nach Geräten suchen (*Discovery/Searching*). SSDP verwendet eine vordefinierte Multicastgruppe, an die die Nachrichten geschickt werden.

Während der anschließenden *Description*-Phase können Control-Points die Beschreibungen von Geräten und Diensten beziehen. Die Beschreibungen sind vorlagenbasierte XML-Dokumente. Sie enthalten gerätespezifische Informationen (Modellname, Hersteller usw.) sowie Methoden (*Action*) und Zustandsvariablen (*State Variable*) der Dienste.

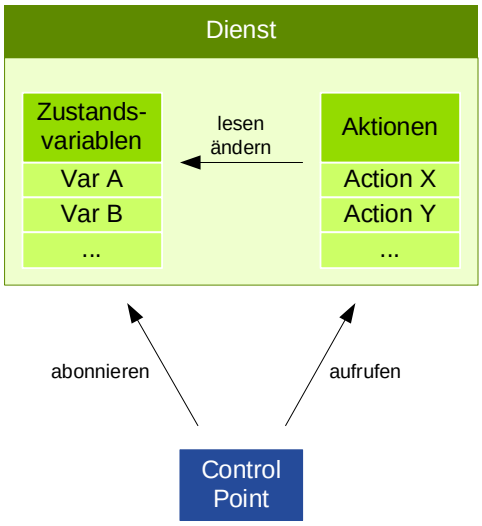

Abbildung 2.16: Beziehungen zwischen Control-Point, Dienst, Aktionen und Zustandsvariablen

Wie der Name bereits andeutet, speichert ein Dienst seinen Kontext in Zustandsvariablen. Diese werden explizit in der Dienstbeschreibung aufgelistet und besitzen jeweils einen Datentyp. Zustandsvariablen können durch Methodenaufrufe verändert – die Parameter der Methoden re-

ferenzieren in diesem Fall die Zustandsvariablen – und in der *Eventing*-Phase durch Control-Points abonniert werden (Abbildung 2.16). Hierfür verwendet UPnP die *Generic Event Notification Architecture* (GENA) [63]. GENA ist wie SSDP im Wesentlichen eine HTTP-Erweiterung. Wenn sich der Wert einer Zustandsvariablen ändert, wird ein Ereignis ausgelöst und die Variable zusammen mit dem neuen Wert an alle Abonnenten geschickt. Die Steuerung der Dienste geschieht in der *Control*-Phase mittels RPC-Aufrufen, wobei UPnP das SOAP-Protokoll einsetzt.

Geräte können optional eine spezielle URL anbieten (als Teil ihrer Beschreibung), mit der sie und ihre Dienste über eine HTML-basierte Schnittstelle gesteuert werden können. Control-Points können diese URL beispielsweise in einem Webbrowser laden und die HTML-Seite dem Benutzer als zusätzliche Möglichkeit anbieten, das Gerät zu bedienen.

Das UPnP-Forum hat inzwischen mehr als zehn DCPs für Geräte und Dienste standardisiert, die von über 200 zertifizierten auf dem Markt erhältlichen Geräten umgesetzt worden sind (Stand 2008). Die meisten Geräte implementieren einen der *MediaServer*, *MediaRenderer* oder *Internet Gateway* Standards, einige wenige den *Printer* Standard. Für andere DCPs wie *Lighting Controls* oder *HVAC* sind bisher keine Geräte entwickelt worden.

2.8 Devices Profile for Web Services

Das Web-Services-Framework ist eine lose Sammlung dezentral spezifizierter Protokolle, die so entworfen sind, dass aus ihnen Spezifikationen für konkrete Anwendungsfälle oder -domänen erstellt werden können – sogenannte Profile. Neben dem *Basic Profile* gibt es auch noch das *Devices Profile for Web Services* (DPWS) [64]. DPWS wurde als Profil über mehrere Web-Services-Spezifikationen mit dem Ziel entworfen, ein serviceorientiertes Framework für ressourcenarme Geräte anzubieten, die fähig sind, sich in einer sicheren lokalen Umgebung gegenseitig zu entdecken, ihre Dienste zu verwenden, Ereignisse zu abonnieren und zu versenden.

DPWS ist ein relativ junges Protokoll (erste Version 2004) und wurde hauptsächlich von Microsoft, Intel und Lexmark spezifiziert. Ursprünglich war DPWS mit dem Untertitel „A Proposal for UPnP 2.0 Device Architecture“ versehen, was zunächst auf eine Nachfolgeversion von UPnP schließen ließ. DPWS wird inzwischen jedoch als eigenständige Technologie dargestellt und ist auch nicht kompatibel zu UPnP. Den Protokollstapel zeigt Abbildung 2.17. DPWS unterscheidet zwischen einem Gerät (*Hosting Service*) und Diensten (*Hosted Services*), von denen ein Gerät beliebig viele anbieten bzw. verwalten kann. Ein Gerät wird selbst als Dienst abgebildet, welcher Nachrichten empfangen und senden kann.

DPWS verwendet für die gegenseitige Entdeckung *WS-Discovery*, *WS-MetadataExchange* [65] und *WS-Transfer* [66]. Die letzteren beiden werden für den Austausch der Geräte- und Dienst-

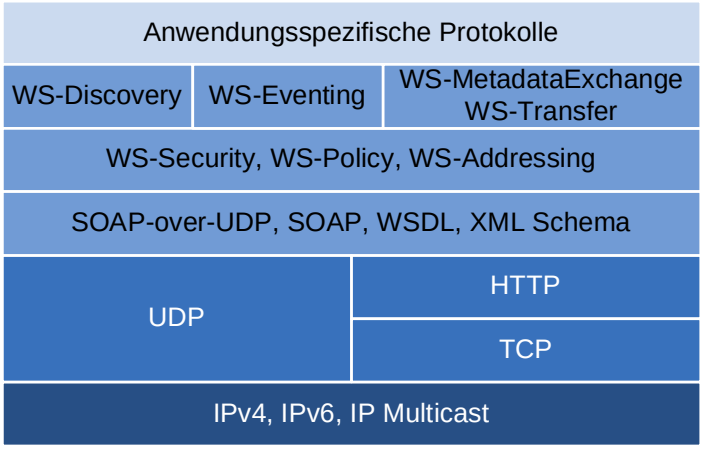

Abbildung 2.17: DPWS-Protokollstapel

beschreibungen eingesetzt, können aber als Teil der Discovery-Phase betrachtet werden.

DPWS beschränkt Discovery auf Geräte, wodurch nur Geräte Target-Services sind, nicht jedoch deren Dienste. Dadurch wird der Netzwerkverkehr reduziert, gleichzeitig jedoch der Discovery-Prozess verlängert (Abbildung 2.18). Zunächst ermittelt ein Dienstnutzer die Geräte in einem Netzwerk entweder passiv durch *Hello*- oder aktiv durch *Probe*-Nachrichten. Dabei kann wieder auf die Suche nach Typen zurückgegriffen werden. Sie bezieht sich jedoch nur auf die Typen, die das Gerät unterstützt, nicht auf die der Dienste. Nach einem *Resolve* ist der Dienstnutzer im Besitz der physikalischen Geräteadresse. Die logische Geräteadresse muss Reboots überstehen können und für die Lebensdauer des Gerätes konstant bleiben. Um an die vom Gerät verwalteten Dienste zu gelangen, schickt der Dienstnutzer eine *Get*-Nachricht (WS-Transfer) an das Gerät. *Get* schickt die XML-Repräsentation einer Web-Services-Ressource, in diesem Fall die des Gerätes, zurück. DPWS verwendet für die Geräterepräsentation das generische *MetadataSection*-Element der WS-MetadataExchange Spezifikation. Der Inhalt dieser Sektionen (u. a. die Elemente *ThisModel* und *ThisDevice*) enthält Angaben zum Hersteller, Modell, Firmware-Version usw. und Referenzen zu den verwalteten Diensten. Über die Endpunkte der Dienste (Dienste werden über das *Hosted*-Element referenziert) kann der Dienstnutzer abschließend mit einem weiteren *Get* die Repräsentation eines Dienstes abfragen. Diese enthält mindestens die zugehörige WSDL.

Geräte- und Dienst-Discovery (funktionales Discovery) sind in DPWS getrennt und nur durch mehrere Schritte ausführbar. Im längsten Fall sind dazu vier Kommunikationsschritte, für mehr als einen verwalteten Dienst weitere Schritte erforderlich: *Hello*/*Probe*, *Resolve*, *Get* an das

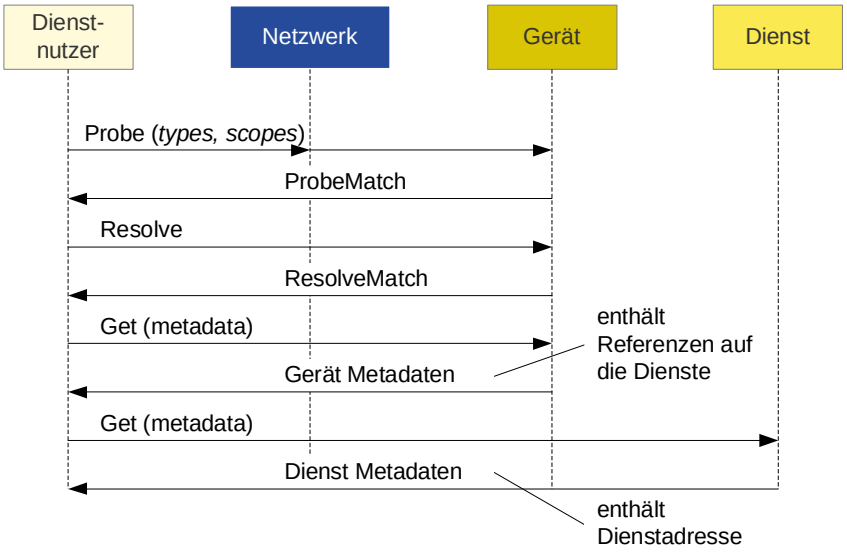

Abbildung 2.18: Ablauf des DPWS-Discovery

Gerät, *Get* an jeden Dienst.

DPWS verwendet für das Abonnieren und Versenden von Ereignis-Nachrichten WS-Eventing, welches weitestgehend so übernommen wurde. Die Dienste sind angewiesen, den *Push*-Mechanismus zu unterstützen, der ebenfalls in WS-Eventing definiert wurde. Um den Anforderungen an ressourcenarme Geräte gerecht zu werden, definiert DPWS den *Action Filter*, der in *Subscription*-Nachrichten verwendet werden kann, um das Abonnement lediglich auf eine Submenge der spezifizierten Operationen zu beschränken.

3 Stand der Wissenschaft und Technik

Schwerpunkte dieser Arbeit sind Konzeption, Umsetzung sowie Anwendung von Frameworks, die Merkmale serviceorientierter Architekturen (SOA) aufweisen und u. a. explizit Geräte integrieren sowie die grafische Anwendungs- bzw. Prozessmodellierung (Abs. 1.5). Für die Frameworks und die grafischen Modellierungswerkzeuge sollen daher im Folgenden der Stand der Wissenschaft und Technik beleuchtet werden. Da in Kapitel 5 der SOA-basierte Aufbau einer Lokalisierungsplattform erläutert wird, sollen außerdem einige Lokalisierungsdienste betrachtet werden, die ebenfalls von Lokalisierungssystemen abstrahierende Schnittstellen besitzen.

3.1 Bussysteme und Frameworks

Viele Bussysteme und Geräte vernetzende Frameworks existierten bereits vor der Erfindung des *World Wide Webs* (WWW). Sie sind in der Regel nicht internetweit einsetzbar und besitzen auch keinen *universellen* Charakter. Da in diesen Bereichen jedoch zunehmend die Anforderungen in Richtung Zusammenführung mit Internettechnologien gehen, werden hier einige dieser Technologien genannt.

Aus den hohen Anforderungen im industriellen Bereich sind verschiedene Bussysteme und Technologien hervorgegangen, die vor allem Echtzeit ermöglichen und eine hohe Robustheit aufweisen.

Das *Local Operating Network* (LON, auch: *LonWorks*) [67] ist ein weltweit eingesetzter Standard zur Automatisierung von Gebäuden und Industrieanlagen. LON-Geräte lassen sich in beliebiger Anzahl und beliebiger Topologie miteinander verbinden, wodurch sich der Verkabelungsaufwand gering halten lässt. Die LON-Spezifikation umfasst alle sieben Schichten des OSI-Modells, wobei die physikalische Schicht über unterschiedliche Medien realisiert werden kann. Es existieren integrierte Chips, die den Stack implementieren. LON ist ein, vor allem in den USA, etablierter Standard, dessen Fokus ausschließlich auf eingebetteten Geräten liegt.

In Europa – besonders in Deutschland – nimmt der *Europäische Installationsbus* (EIB) [68] eine gesonderte Stellung ein. Es gibt mehrere tausend EIB-fähige Geräte für die Gebäudeautomatisierung auf dem Markt. Inzwischen werden auch immer öfter Fertighäuser mit EIB ausgestattet, da sich eine solche Installation langfristig rechnet (Stromeinsparung, Montageaufwand bei Änderungen am Haus usw.). EIB ist ein offener Standard, jedoch verfügt er mit nur 9,6 kBit/s

über eine sehr niedrige Datenübertragungsrate.

Der Feldbus *PROFIBUS* [69] ist Weltmarktführer im Bereich der Fertigungsautomatisierung und Prozessautomatisierung. Er deckt die untersten beiden Schichten des OSI-Modells ab und definiert auf Anwendungsebene zyklischen, azyklischen und isochronen Datenaustausch.

Im Fahrzeugbereich haben sich vor allem das *Controller Area Network* (CAN) [70] und das kostengünstige und für einfache Kommunikationsaufgaben gedachte *Local Interconnect Network* (LIN) [71] durchgesetzt. CAN ist ein Bussystem, welches in sicherheitskritischen Anwendungsgebieten, die eine hohe Datensicherheit erfordern, zum Einsatz kommt. Dieses sind vor allem die Automobil-, Medizin- und Flugzeugtechnik. Gegenwärtige Anforderungen im Automobilbau (steigende Anzahl an Sensoren, Aktuatoren, Multimediafähigkeit usw.) verlangen nach mehr Flexibilität und vor allem höheren Datenraten. Der relativ junge *FlexRay*-Bus [72, 73] adressiert genau diese Anforderungen.

Das rein auf Geräte (v. a. Drucker und Scanner) zielende Framework *Salutation* wurde seit 1995 innerhalb des Salutation-Konsortiums entwickelt. Salutation ist vollständig unabhängig von jeglichen Plattformen und Transportprotokollen [74]. Den gemeinsamen Nenner bildet die Datennotation, die mittels der ASN.1 (Abstract Syntax Notation One) [75] realisiert wird. Salutation kann als *universell* bezeichnet werden, jedoch liegt der Fokus ausschließlich auf Geräten. Salutation schaffte es mit einigen Geräten auf den Markt zu gelangen, ein Marktdurchbruch konnte jedoch nie erzielt werden. 2005 stellte das Konsortium seine Arbeit ein und löste sich auf.

Home Audio Video Interoperability (HAVi) [76] ist ein Standard zur interoperablen Vernetzung von Audio/Video-Geräten (AV) wie Tuner, TV, Kameras usw. Er basiert vollständig auf dem IEEE-1394-Bus (FireWire), welchen er für den Transport von Echtzeit-AV-Streams benötigt. HAVi adressiert daher vor allem den Heimbereich. Es gibt eine Java-API, welche zur Entwicklung von HAVi-fähigen Anwendungen verwendet werden kann [77]. Für den Einsatz von HAVi werden jedoch Gebühren fällig.

Universell einsetzbare und vernetzende Frameworks, die auch Geräte einbeziehen, sollten vor allem frei von Abhängigkeiten von Programmiersprachen, Hardwareplattformen und Betriebssystemen sein. Nur so kann eine breite Akzeptanz erreicht werden.

Vor allem Java bietet als plattformunabhängige Programmiersprache sehr gute Voraussetzungen hinsichtlich einer breiten Anwendbarkeit. Die Abhängigkeit zur Programmiersprache bleibt jedoch bestehen. Das von Sun entwickelte, auf Java basierende Jini [78] gilt als eines der ersten Frameworks, das viele Merkmale einer serviceorientierten Architektur umsetzt.

In einer Jini-Umgebung können heterogene Geräte, deren Dienste und Dienstnutzer konfigurations- und wartungsfrei in einer spontanen Art und Weise eingebracht werden, ohne zuvor die

existierende Netzwerk- und Dienstinfrastruktur kennen zu müssen. Die Schnittstellenbeschreibung erfolgt in Form von Java-Interfaces, die ein Dienstanbieter bei einem zentralen *Lookup-Service* (Registrierung) zusammen mit einem *Service-Object* (Dienstobjekt), welches die Schnittstelle implementiert, hinterlegen muss. Dienstnutzer durchsuchen den Lookup-Service und verwenden Dienste, indem sie eine Kopie des registrierten Dienstobjektes herunterladen. Die Kommunikation erfolgt für den Dienstnutzer transparent (in der Regel über RMI), da dieser das Dienstobjekt lediglich über die Schnittstelle verwendet. Dienstobjekte können daher beispielsweise auch über SOAP oder CORBA kommunizieren. Jini enthält die *Remote Event API* für verteilte Ereignisse und die *Java Transaction API* (JTA) für die Koordination verteilter Transaktionen. Jini gelang nie ein nennenswerter Durchbruch.

Die *Open Service Gateway Initiative* (OSGi) realisiert wie Jini eine auf der *Java Virtual Machine* (JVM) basierende, serviceorientierte Architektur. OSGi erweitert dazu die Java-Plattform um zusätzliche Funktionalitäten, so dass ein hochdynamisches Framework entsteht, welches das Hinzufügen, Aktualisieren und Entfernen von Diensten sowie das Auflösen aller Abhängigkeiten zur Laufzeit innerhalb *einer einzigen JVM* realisiert. Um die Anforderungen an eingebettete Systeme besser abzudecken, erfordert die minimale Ausführungsumgebung nur eine Untermenge der *Java Micro Edition* (J2ME) *Connected Limited Device Configuration* (CDC/CLDC) Profile. OSGi wird seit 1999 von der OSGi-Alliance [79], bestehend aus verschiedenen Firmen wie IBM, Sun Microsystems, Nokia, Samsung, Siemens und Motorola, entwickelt und wurde seither in mehreren Generationen (aktuell R4) spezifiziert.

OSGi war ursprünglich für die Verwendung auf eingebetteten Geräten gedacht, anfangs im Speziellen für Digitalempfänger und Modems. Mit der Zeit erschlossen sich jedoch zunehmend andere Anwendungsgebiete und so findet man OSGi-Plattformen heute im Automobilbau, in der Haus- und Gebäudeautomatisierung, in der Telekommunikation und im Energiemanagement-Sektor [80]. Einen völlig neuen Einsatz findet OSGi in reinen Softwareanwendungen wie beispielsweise Eclipse [81].

Die OSGi-Alliance stellt bereits viele standardisierte Dienste als Java-Interface-Bibliotheken zur Verfügung, beispielsweise für HTTP-Server, Ereignisdienste, Logging, Administrationsdienste, Brücken zu UPnP- und Jini-Netzwerken oder XML-Dienste (Abbildung 3.1).

Die *Common Object Request Broker Architecture* (CORBA) [82] ist ebenfalls ein plattform- und programmiersprachenunabhängiges Framework, welches die Entwicklung und Nutzung von verteilten Anwendungen in einer objektorientierten Art und Weise erlaubt. Dazu können Dienstschnittstellen mit der *Interface Definition Language* (IDL) definiert werden. Aus der IDL lassen sich Server- und Clientcode generieren. Die Komplexität des Protokolls ist dem Benutzer dadurch verborgen. Die Kommunikation findet in den meisten Fällen über das in CORBA de-

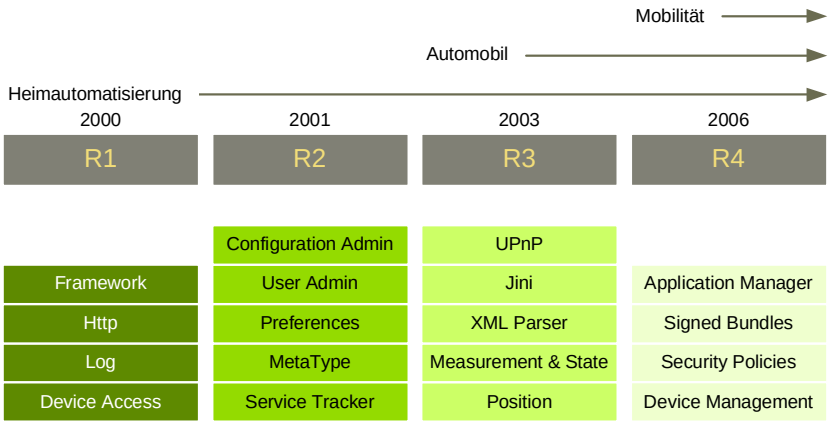

Abbildung 3.1: Standardisierte OSGi-Services und ihre Domänen

finierte *Internet Inter-ORB Protocol* (IIOP) statt. CORBA spezifiziert eine Reihe zusätzlicher Infrastrukturdienste, die Ereignisse, Sicherheit, Transaktionen usw. unterstützen. Diese Dienste sowie die Implementierungen werden durch plattformspezifische *Object Request Broker* (ORB) realisiert.

In [83] wird mit *Eucalyptus* eine Architektur für ressourcenarme, drahtlose Geräte vorgestellt. Die Architektur basiert auf einer objektorientierten Plattform, die ein an CORBA angelehntes Protokoll – den *Koala ORB* – verwendet.

PABADIS und Pini adressieren ebenfalls ressourcenarme Geräte, jedoch im Umfeld der Automatisierung und an Jini angelehnt. Die Lösung bietet eine vernetzte Umgebung an, durch das Fertigungsaufträge selbstständig navigieren [84].

Das bereits 1999 veröffentlichte *MOCA*-Framework [85] unterstützt Dienstsuche und -nutzung auf mobilen Geräten. Dienste werden dazu in Java implementiert und zentral registriert. Mobile Geräte können die Dienste dynamisch entdecken und herunterladen. MOCA stellt einen eigenen *Class Loader* bereit, woduch alle Dienste in einer einzigen JVM laufen. MOCA enthält Konzepte, die später in Jini und OSGi wieder auftauchen.

Ziel des europäischen Forschungsprojektes SIRENA (*Service Infrastructure for Real-time Embedded Networked Devices*) [86] war die plattformunabhängige Vernetzung von eingebetteten Geräten in den vier unterschiedlichen Domänen Industrie, Automobiltechnik, Telekommunikation und Heimautomatisierung. Das SIRENA-Framework basiert auf DPWS und UPnP, spezifiziert einige Erweiterungen zu bestehenden domänenspezifischen Frameworks und definiert Werkzeuge und Methodiken, um Geräte in eine SIRENA-Umgebung zu integrieren [87]. Einige der hier vorgestellten Ergebnisse resultieren aus der Mitarbeit im SIRENA-Konsortium. Unter anderem

wurden für das UPnP-Framework Werkzeuge [88] und Erweiterungen für die Unterstützung von *Quality of Service* [89] für die Audio/Video-Domäne und die Integration von Workflow-Systemen [90] entwickelt. In [91] wird ein grafisches Werkzeug zum Management von UPnP-Geräten vorgestellt. Außerdem wurde demonstriert, wie eine große Zahl von Bluetooth-basierten Geräten energiesparend in eine SOA eingebracht werden kann [92].

Die *Web Services for Devices* (WS4D) Initiative [93] ist ein Zusammenschluß ehemaliger SIRENA-Partner und führt die Ergebnisse des SIRENA-Projektes fort. Innerhalb von WS4D werden weiterhin vor allem DPWS evaluiert [94, 95, 96] und Werkzeuge für die Vernetzung von eingebetteten Geräten entwickelt [97].

Von der Erkenntnis ausgehend, dass die meisten bestehenden Web-Services-Werkzeuge zu groß für eingebettete Systeme sind, wird in [98] ein WS-Toolkit vorgestellt und evaluiert, welches die Anforderungen eingebetteter Geräte adressiert. Die Komponenten folgen allerdings keinem Profil, wie es bei DPWS der Fall ist. Die gleiche Veröffentlichung hebt REST aufgrund seiner Einfachheit als interessante Alternative hervor.

Ein Ansatz, auf Geräte REST-basierend zuzugreifen, wird mit dem *pico-REST*-Protokoll in [99] vorgestellt. Allerdings beschränken sich die Autoren auf die reine Interaktion, weder Discovery noch Eventing werden betrachtet. In [100] schlägt der Autor eine Methodik vor, wie UPnP ohne SOAP auskommen kann, indem Statusvariablen und Aktionen REST-basiert abgebildet werden.

Der Versuch, einen geeigneten internetweiten Ereignismechanismus zu finden, wurde bereits 1998 auf der eigens dafür veranstalteten Konferenz WISEN [101] unternommen (u. a. wurde hier GENA, was später in UPnP verwendet wurde, vorgestellt). Es wurde sich jedoch nie auf eine spezielle Lösung geeinigt. HTTP selbst unterstützt keinen *Publish-Subscribe-Mechanismus*. In [102] wird jedoch eine Realisierung vorgestellt, die HTTP *POST* und einige zusätzliche HTTP-Header verwendet. Für Webseiten haben sich in jüngster Zeit neue Möglichkeiten durch Techniken wie *Comet*[1] [103] eröffnet, die (unendlich) lang dauernde HTTP-Connections ausnutzen. Über diese können Server dann Ereignisse an den Client (z. B. Webbrowser) schicken.

Während die o. g. Frameworks und Forschungsprojekte größtenteils auf die Vernetzung und Nutzung von Geräten und Diensten abzielen, beschäftigen sich andere Arbeiten mit dem Vergleich der Ansätze. Laut [104] liegt die Performance von Web Services unter der von Java RMI und CORBA, dennoch ist die Differenz für realistische Anwendungen vernachlässigbar. In der praktischen Anwendbarkeit liegt CORBA in [105] deutlich vorne, da es alle wichtigen Dienste bereits mitbringt. Im Gegensatz zu CORBA und EJB sind Web Services jedoch wesentlich einfacher und vor allem besser für die Anwendungsintegration geeignet, da sie keine Objektori-

[1] `http://alex.dojotoolkit.org/?p=545`

entierung voraussetzen.

In [106] werden die beiden Frameworks REST/HTTP und WS-* hinsichtlich ihrer Komplexität miteinander verglichen. Dazu werden die Anzahl an zu treffenden Entscheidungen und dazu vorhandenen Alternativen gemessen (z. B. Datenformat, Transportprotokoll, Dienstkomposition usw.). Insgesamt verfolgt WS-* den Ansatz der *Freiheit, wählen zu können* und REST/HTTP den der *Freiheit, nicht wählen zu müssen.* Anstatt einen der beiden Ansätze zu favorisieren, erstellt die Arbeit eine Art Leitfaden, anhand dessen sich die richtige Technologie für eine gegebene Aufgabe finden lässt.

Zusammenfassend kann behauptet werden, dass sich bis heute keines der vorgestellten Frameworks vollständig durchgesetzt hat. Für die meisten Lösungen sind folgende Gründe verantwortlich: Entweder sind die Ansätze an eine Programmiersprache gebunden (OSGi, EJB, Jini) oder an ein bestimmtes Programmierkonzept (RPC/RMI in UPnP und Jini, OO in CORBA), oder aber sie adressieren nur ausgewählte Domänen (CAN, PROFIBUS usw.). Einzig UPnP und DPWS bilden hier eine Ausnahme. Das UPnP-Framework (UDA) weist jedoch mit dem nicht skalierbaren Discovery, dem ebenfalls schlecht skalierbaren Ereignismechanismus und dem fehlenden Sicherheitskonzept erhebliche technische Defizite auf. Die Stärke von UPnP liegt in den existierenden Anwendungsprotokollen (DCP). DPWS unterliegt dem Nachteil der hohen Komplexität und der noch relativ instabilen Spezifikation, kann jedoch aufgrund der Nähe zu den serviceorientierten Architekturen und der Geschäftsinformatik als Framework mit einem hohen Zukunftspotenziel gewertet werden.

3.2 Middleware für Lokalisierungsdienste

Lokalisierungsplattformen sind Systeme, die Positionsdaten von Objekten oder Benutzern messen, aufbereiten und zur Verfügung stellen. Viele dieser Plattformen sind dabei an ein bestimmtes Lokalisierungssystem gebunden. Beispielsweise liefert das Echtzeit-Lokalisierungssystem *Ubisense* [107] eine Plattform aus, in der die gemessenen Objekte in 3D dargestellt werden können. Und mit der Ubisense-API steht eine auf C++ basierende Schnittstelle zur Verfügung.

Lokalisierungsdienste, die von Lokalisierungssystemen abstrahieren, müssen über Schnittstellen verfügen, die die darunterliegenden Systeme homogen abbilden. Davon werden im Folgenden einige kommerzielle und akademische Ansätze vorgestellt.

Das *Mobile Location Protocol* (MLP) der *Open Mobile Alliance* (OMA) [108] definiert die Schnittstelle zwischen einem *Location Server* und einem *Location Services Client* und wird vor allem im Telekommunikationsbereich eingesetzt. Die Schnittstelle besteht aus fünf Operationen und ermöglicht die Positionsabfrage von mobilen Nutzern bzw. das Abonnieren für fortlaufende

Positionsinformationen. MLP kommuniziert über den Austausch von XML-Nachrichten unabhängig vom verwendeten Transport, basiert jedoch nicht auf den standardisierten Web Services. Außerdem können beliebige Koordinatensysteme verwendet werden. Alle Positionen in Form von Punkten oder geometrischen Flächen/Körpern referenzieren ihr Koordinatensystem mittels einer URI.

Die Java-Plattform verfügt mit der JSR 179 (*Java Specification Request*) und der JSR 293 über zwei *Location APIs*, die von Anwendungen verwendet werden können, die die aktuelle Position des Gerätes, auf dem sie laufen, benötigen. Die APIs (JSR 293 erweitert JSR 179) verstecken die eigentliche Positionsermittlung bzw. den verwendeten Anbieter vor den Anwendungen. Koordinaten werden als Latitude-Longitude-Altitude-Datensatz im WGS84-Format (GPS) angegeben.

TraX ist ein Middleware-Framework, welches verschiedene Schnittstellen wie MLP und Java Location API vereint und dabei besonderen Fokus auf die Sicherheit der Daten legt [109].

Place Lab [110, 111] erlaubt Geräten wie Mobiltelefonen oder Laptops die eigene Positionsermittlung, indem sie die sie umgebenen Beacons wie WLAN Access Points oder andere fest positionierte Beacons abhören. Über deren IDs (z. B. MAC-Adresse) kann über eine Datenbank die Position geschätzt werden. Place Lab verwendet als Schnittstelle die o. g. Java Location API.

Einen ähnlichen Ansatz wie Place Lab verfolgt das *PLIM*-Framework (*Presence, Location and Instant Messaging*) [112], welches die Eigenlokalisierung von Geräten erlaubt. Hierzu wurde im Rahmen des Forschungsprojektes *GigaMobile* eine generische Lokalisierungsschnittstelle in Form einer Software-API entwickelt, die ebenfalls Bluetooth, WLAN usw. für den Benutzer transparent verwendet [113].

MiddleWhere [114] abstrahiert von unterschiedlichen Lokalisierungssystemen und stellt seinen Lokalisierungsdienst mittels CORBA zur Verfügung.

Die im Rahmen eines Sonderforschungsbereiches an der Universität Stuttgart entwickelte Nexus-Plattform bildet die Umgebung (Welt) als Modell ab und speichert darin Positionen von realen und virtuellen Objekten. Die Plattform verwaltet die Informationen verteilt in *Context Server* und stellt sie mittels *Nexus Nodes* kontextabhängigen Anwendungen zur Verfügung [115]. Auf die Daten ist über ein auf RMI basierendes Framework zugreifbar [116]. Ein entsprechender generischer skalierbarer Lokalisierungsdienst wird in [117] vorgestellt, der auf einer hierarchischen verteilten Architektur basiert und für eine sehr große Anzahl von mobilen Objekten und Dienstnutzern geeignet ist.

In [118] werden verschiedene Architekturen und Lösungen vorgestellt, mit deren Hilfe ein Lokalisierungsdienst umgesetzt werden kann. Der Autor beschreibt dazu abstrakte Konzepte, die unabhängig von spezifischen Lokalisierungssystemen gelten.

Die vorgestellten Middlewarelösungen für Lokalisierungsdienste sind entweder spezifisch für bestimmte Lokalisierungssysteme, stellen eine API für eine bestimmte Programmiersprache dar oder verwenden proprietäre Technologien. In dieser Arbeit wird eine serviceorientierte Lokalisierungsplattform aufgebaut, die auf den standardisierten Web Services und DPWS setzt.

3.3 Grafische Prozess-Modellierungswerkzeuge für Daten, Dienste und Geräte

Damit der Ablauf komplexer Prozesse, Workflows oder Anwendungen besser abgebildet und leicht an Änderungen angepasst werden kann, bedient man sich auf Designebene oft grafischer Modellierungswerkzeuge. Diese können in der Regel auch von weniger technisch versierten Benutzern bedient werden und erlauben gleichzeitig einen umfassenden visuellen Einblick in das Ziel der Anwendung.

Im Bereich der Web Services können mit Hilfe der *Business Process Execution Language* (BPEL) Geschäftsprozesse definiert und von einer BPEL-Engine ausgeführt werden [119]. Kommerzielle Werkzeuge wie der *Oracle BPEL Process Manager* [120], der *IBM WebSphere Business Modeler* [121] oder der *BizTalk Server 2006 Orchestration Designer for Business Analysts* für Microsoft Visio helfen bei der Modellierung. Sie enthalten in der Regel grafische Elemente für Web-Services-Aufrufe, Datentransformationen und Kontrollelemente wie Schleifen, Sequenzen und Bedingungen usw. Prozesse werden als Graphen abgebildet. Vergleichbare frei verfügbare Werkzeuge gibt es mit dem *Eclipse BPEL Project* [122] oder dem *Active BPEL Designer* [123].

BPEL-Modellierungswerkzeuge basieren auf den abstrakten Definitionen von WSDL-Dokumenten. Die zu verwendenden Dienste sind demnach auf Web Services beschränkt.

Die *OutSystems Plattform* [124] erlaubt ebenfalls die Erstellung von Geschäftsanwendungen auf grafischer Ebene. Die codefreien Programme werden im *OutSystems Service Studio* erzeugt und können dann wahlweise auf .NET- oder Java-Systemen installiert werden. Die Plattform ist nicht auf die Verwendung von Web Services beschränkt, es können ebenso Mail-Server, Webseiten oder Datenbanken angekoppelt werden [125].

LabVIEW [126] ist ein grafisches Programmierwerkzeug, mit welchem sehr einfach parallel ausführbare Programme erstellt werden können. Funktionsblöcke werden dazu mit Drähten verbunden, durch welche anschließend Daten verschickt werden (Datenfluss-Prinzip). Funktionsblöcke können außerdem verschachtelt werden. Ein Funktionsblock beginnt seine Ausführung, wenn Daten an allen Drähten anliegen und sendet anschließend die Ergebnisse über seine Ausgänge. LabVIEW ist ein sehr umfangreiches und ausgereiftes System, welches vor allem für die Datenaggregation und Datenanalyse im Laborumfeld eingesetzt wird. Sowohl die Entwicklungs-

als auch die Ausführungsumgebung müssen kommerziell erworben werden.

Im Desktopbereich gibt es ebenfalls grafische Werkzeuge, mit denen sich neue Anwendungen oder Prozesse kreieren lassen. *Particls* [127] ist eine Überwachungsplattform für den Einsatz auf dem persönlichen Desktop. Es stehen verschiedene *Input Adapter* wie RSS Feed Reader, MS Money Monitor, Ebay Auktionstracker, TV-Listings usw. zur Verfügung. Diese können mit Filtern und Bedingungen versehen werden, und der Benutzer wird in der Folge von Particls informiert, wenn bestimmte Ereignisse eintreten. Als *Output Adapter* gibt es beispielsweise SMS und E-Mail, Tickerkomponenten oder Popups.

Mac-OSX-Betriebssysteme enthalten den *Automator*. Mit diesem Werkzeug lassen sich u. a. Web Services, das lokale Dateisystem und die Apple-spezifischen Programme wie iCal, Quick Time, Mail, PDF usw. benutzen und automatisieren. Dazu stehen umfangreiche grafische Dialoge zur Verfügung.

Yahoo Pipes [128] ist ein browserbasiertes Modellierungswerkzeug, um Daten aus dem WWW zu filtern, zu aggregieren und zu kombinieren und schließlich als neue Datenquelle zur Verfügung zu stellen. Yahoo Pipes sind sequenziell ablaufende Datenfluss-Anwendungen. Als Datenquellen fungieren HTML-Seiten, RSS-Daten und Dienste, die XML/JSON ausgeben.

Einen ähnlichen Ansatz verfolgt *Macro.scopia* [129]. Hier lassen sich ebenfalls in einem Webbrowser *Macros* modellieren, die aus Blöcken bestehen und in Webseiten eingebettet werden können, wo sie dann ausgeführt werden. Wie in Yahoo Pipes sind die Blöcke jedoch von der Plattform vorgegeben und können nicht erweitert werden. Als Datenquellen können u. a. RSS-Daten und Google-Kalender oder -Dokumente verwendet werden. Ein Macro endet mit einem Visualierungsblock. Macro.scopia verfügt über einige solcher Blöcke, die für unterschiedliche Visualisierungen strukturierter Daten geeignet sind.

Die in dieser Arbeit vorgestellte Pipes-Plattform vereint die Charakteristiken verschiedener grafischer Anwendungsmodellierer: Sie ist nicht auf ein bestimmtes Framework beschränkt, sondern offen für jedes Bussystem bzw. jede Technologie (LabVIEW). Die Modellierung findet in einem Webbrowser und nicht in einer Desktopanwendung statt (Yahoo Pipes und Macro.scopia). Außerdem kann sie durch benutzerspezifische Funktionsblöcke (*Module*) erweitert werden (LabVIEW, OutSystems) und ihre Module sind verschachtelbar (LabVIEW).

4 Anforderungen an universelle, geräteintegrierende Frameworks

Im Folgenden werden die vielfältigen Anforderungen erläutert, die universell anwendbare und geräteintegrierende Frameworks abdecken sollten. Das Kapitel dient als informative Grundlage, um unterschiedliche Frameworks bewerten zu können. Gleichzeitig kann es für den Entwurf neuer Frameworks verwendet werden. Neben allgemeinen Anforderungen werden Anforderungen an Basisdienste, Werkzeuge, Netzwerk und Kommunikation beschrieben.

4.1 Allgemeine Anforderungen

4.1.1 Abstraktion

An ein Framework, welches domänenübergreifend funktionieren soll, sind viele Anforderungen gestellt, die sich zum Teil von den Anforderungen, denen abgeschlossene proprietäre Lösungen zugrunde liegen, unterscheiden. Um die unterschiedlichen Anwendungsgebiete und Gerätetypen zu adressieren, und um eine Interaktion zwischen ihnen zu ermöglichen, muss die Lösung adaptierbar sein, also einen gewissen Abstraktionsgrad aufweisen. So sollte sich das Framework nicht auf eine spezifische Domäne beschränken. Statt dessen soll es sowohl in der Industrie, in der Haus- und Gebäudeautomatisierung, in der Telekommunikation als auch im Fahrzeugbau eingesetzt werden können. Das Framework muss weiterhin frei von Abhängigkeiten zu bestimmten Hardwareplattformen, zu Betriebssystemen und zu Programmiersprachen sein.

4.1.2 Geräte- und Dienstevielfalt

Das Framework soll eine möglichst breite Vielfalt an Gerätetypen unterstützen, von kleinsten Geräten im 8-Bit-Prozessorbereich bis hin zu intelligenten Geräten wie PDAs, Telefonen oder eben PCs. Diese Anforderung dürfte umso leichter zu erfüllen sein, als dass die Leistungsfähigkeit der eingebetteten Systeme kontinuierlich steigt. Dennoch gilt es, die Parameter wie Ressourcen, Energieverbrauch und Kosten klein zu halten. Viele mobile Geräte sind auf Akkus angewiesen.

Weiterhin darf die Anzahl der in das Framework zu integrierenden Geräte und Dienste nach oben hin nicht begrenzt werden. Unter Umständen und je nach Anwendungsgebiet kann diese Anforderung durch Quality-of-Service-Attribute (z. B. zusätzliche Performanceanforderungen, begrenzte Bandbreite usw.) später zur Laufzeit eingeschränkt werden. Diese Einschränkung soll

ihre Ursache jedoch in der Anwendung und nicht im Framework haben.

4.1.3 Systemeigenschaften

Das Framework sollte *skalierbar*, *robust* und *erweiterbar* sein. *Skalierbarkeit* bezeichnet die dabei Fähigkeit eines Systems, (beliebig viele) Komponenten zur Architektur unter Aufrechterhaltung der funktionalen und nicht-funktionalen Systemeigenschaften hinzuzufügen. Skalierbarkeit wird in der Regel durch dezentrale Ansätze realisiert, da zentrale Komponenten zu „Flaschenhälsen“ führen, die die Skalierbarkeit begrenzen. *Zuverlässigkeit/Robustheit* bezeichnet den Grad eines Systems hinsichtlich der Sicherheit gegen technisches und menschliches Versagen. Ein System ist umso zuverlässiger, je unanfälliger es gegen Ausfälle und Bedienungsfehler ist. Zuverlässigkeit kann insbesondere durch Schaffen von Redundanzen und Dezentralismus erreicht werden. Redundanzen wiederum unterliegen der Problematik der Konsistenzerhaltung durch die notwendige Replikation. *Erweiterbarkeit* bezeichnet die Fähigkeit eines Systems, es um zusätzliche Fähigkeiten/Komponenten zu ergänzen.

4.2 Basisdienste, Komponenten und Werkzeuge

4.2.1 Plug-and-Play

Plug-and-Play bezeichnet die Fähigkeit eines Systems, das Hinzufügen und Entfernen neuer bzw. vorhandener Komponenten selbstständig zu erkennen. Der Begriff stammt ursprünglich aus dem Hardwarebereich [130]. Ein flexibles und leistungsfähiges, geräteintegrierendes Framework benötigt jedoch zusätzlich Plug-and-Play auf der funktionalen Ebene (Softwareebene).

Im Einzelnen müssen folgende Anforderungen abgedeckt werden:

- Die Umgebung erkennt das Hinzufügen einer neuen Komponente zur Laufzeit automatisch, kann es identifizieren und ist in der Lage, alle Schritte automatisch auszuführen, die notwendig sind, um mit der Komponente zu kommunizieren.
- Die Umgebung erkennt ebenso wie das Hinzufügen auch das Entfernen der Komponente. Das Entfernen der Komponente, sei es beabsichtigt oder unvorhergesehen, darf unter keinen Umständen zu einem Ausfall des Gesamtsystems führen.
- Die Komponente erkennt die Umgebung (*symmetrisches* Plug-and-Play). Dadurch können Komponenten gleichzeitig als aktive und passive Kommunikationspartner auftreten, wodurch wesentlich flexiblere Anwendungen realisierbar sind, als dieses bei einem zentralisierten Prinzip der Fall wäre, in welchem nur der zentrale *Master* aktiv ist und sich alle anderen Komponenten (*Slaves*) passiv verhalten.

- Der Plug-and-Play-Mechanismus sollte auch ohne zentrale Komponenten funktionieren. Zentrale Komponenten können die Robustheit einer Architektur verringern, wenn das System auf sie angewiesen ist. Fällt eine solche aus oder kommt es zu Kapazitäts- oder Performance-Engpässen, kann unter Umständen das gesamte System nicht mehr richtig arbeiten. Aus letzterem Grund ist auch die Skalierbarkeit einer auf zentralen Komponenten basierenden Architektur gefährdet. Weiterhin kann die Installation einer zentralen Komponente in mobilen Ad-hoc-Umgebungen und spontan aufgebauten Netzen nicht vorausgesetzt werden, wodurch ihre Notwendigkeit für Geräte ungeeignet ist. Auf der anderen Seite können zentrale Komponenten wie Verzeichnisdienste redundanten Datenaustausch verhindern und somit zu einer Verringerung des Netzwerkverkehrs beitragen. Im Idealfall ist eine Plug-and-Play-Lösung unabhängig von zentralen Komponenten, kann diese aber zusätzlich optional zur Laufzeit einbringen, um den genannten Vorteil auszunutzen.

4.2.2 Beschreibung

Das Framework muss einen Mechanismus bereitstellen, um die Geräte und Dienste auf Anwendungsebene spezifizieren zu können. Mit formalen Beschreibungen lässt sich beispielsweise automatisch Code erzeugen. Weiterhin können sie zur Laufzeit verwendet werden, um Dienste und Geräte dynamisch auf Eigenschaften hin zu inspizieren. Die Beschreibung umfasst dabei alle relevanten Daten eines Gerätes wie gerätespezifische Parameter und Schnittstellen in einer deklarativen Form. Der Beschreibungsmechanismus muss uniform, domänenübergreifend einsetzbar und unabhängig von Plattformen, physikalischen Ressourcen und Programmiersprachen sein. Er muß sowohl für Kleinstgeräte wie Sensoren als auch für hochintelligente Geräte und Softwarekomponenten im Allgemeinen anwendbar sein. Im Idealfall ist die Sprache sowohl maschinenlesbar, also durch Softwarewerkzeuge verarbeitbar, als auch für Menschen verständlich. Normalerweise erfüllen textbasierte Auszeichnungssprachen wie XML diese Anforderung. Durch den Einsatz einer solchen deklarativen, unabhängigen Beschreibungssprache wird eine breite Akzeptanz ermöglicht.

Die Beschreibung sollte unabhängig von der Implementierung sein und nur die Angaben enthalten, die benötigt werden, damit eine Anwendung mit dem Dienst kommunizieren kann. Indem von der Implementierung abstrahiert wird, sind Anwendungen stärker von den Komponenten entkoppelt, da sie lediglich die Schnittstellen kennen, nicht jedoch, wie diese umgesetzt sind. Dadurch ist die Möglichkeit gegeben, die Implementierung später zu ändern, ohne die darauf zugreifenden Anwendungen anpassen zu müssen – eine notwendige Voraussetzung dafür, dass Softwareaktualisierungen auch zur Laufzeit durchgeführt werden können, ohne dass in Verbindung stehende Anwendungen davon betroffen sind. Außerdem können Geräte unterschiedlicher

Hersteller eingesetzt werden (oder später gegeneinander ausgetauscht werden), solange sie die gleiche Schnittstelle verwenden.

4.2.3 Ereignisse

Wenn Wasser anfängt zu kochen, wenn eine Prozedur eine Berechnung beendet, wenn ein neuer Tag beginnt oder wenn ein Benutzer die linke Maustaste drückt, dann sind das alles Ereignisse. Ereignisse sind nicht an die Domäne der Informatik gebunden, sondern können überall und jederzeit auftreten, in der Natur, in einem Gerät oder in einem Stück Software. Vereinfacht ausgedrückt kann jede Zustandsänderung in einem System als Ereignis aufgefasst werden. In der Informationstechnik ist man oftmals am Auftreten bestimmter Ereignisse interessiert. Man kann zwischen physikalischen und logischen Ereignissen unterscheiden. Für Geräte und insbesondere Sensoren spielen physikalische Ereignisse wie der Bruch einer Glasscheibe oder das Übersteigen einer bestimmten Temperatur eine Rolle, in der Software sind es dagegen logische Ereignisse wie das Eintreffen einer neuen E-Mail.

Ereignisse treten einfach auf, unabhängig davon, ob sie überwacht werden oder ob jemand über sie benachrichtigt wird. Um Ereignisse auswerten und auf sie reagieren zu können, müssen verteilte Architekturen eine ereignisbasierte Middleware zur Verfügung stellen. Als *Produzenten* werden die Quellen bezeichnet, die die Ereignisse identifizieren und sie als Ereignisnachrichten veröffentlichen (*publish*). *Konsumenten* dagegen sind die Interessenten, die Ereignisnachrichten empfangen und weiter verarbeiten. Dazu müssen sie in der Regel den Empfang von Ereignisnachrichten abonnieren (*subscribe*). In der Folge wird dieses Muster *Publish-Subscribe-Pattern* genannt.

Laut der in [131] vorgestellten Taxonomie lassen sich die potenziellen Eigenschaften einer bestimmten verteilten ereignisbasierenden Middleware in die drei Säulen *Ereignismodell*, *Ereignisdienst* (die das Modell implementierende und für das Ereignissystem als Middleware zur Verfügung stehende Komponente) und *Ereignissystem* (die Anwendung, die den Ereignisdienst nutzt) klassifizieren. Ereignismodelle beschreiben das auf Ereignissen basierende Kommunikationsmodell, beispielsweise *Peer-to-Peer*, *Vermittler* (auch *Broker Service*) oder *implizites* Modell (Abbildung 4.1). Im Peer-to-Peer-Modell kommunizieren Produzent und Konsument direkt. Beim Vermittler-Modell dagegen wird das Abonnieren und Veröffentlichen durch eine dritte Komponente realisiert. Beim impliziten Modell schließlich werden Ereignistypen abonniert – für den Konsumenten ist es nicht ersichtlich von welchen Produzenten die Nachrichten kommen, es können sogar verschiedene sein.

Für geräteintegrierende Frameworks sind vor allem das Peer-to-Peer- und das Vermittler-Modell von Bedeutung. Peer-to-Peer deshalb, weil die Voraussetzung einer zentralen Kompo-

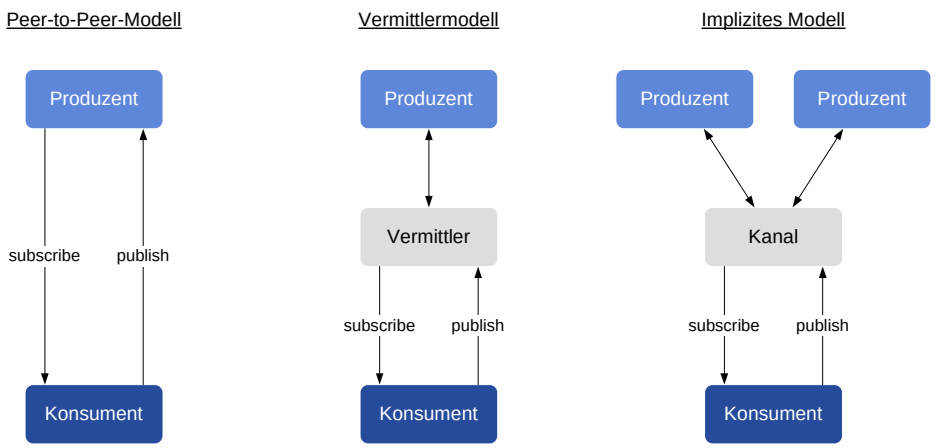

Abbildung 4.1: Grundlegende Ereignismodelle in ereignisbasierten Anwendungen

nente umgangen wird, und das Vermittler-Modell wird für Ereignisproduzenten mit Ressourcenbeschränkung wie Sensoren benötigt, erstens, um überhaupt Ereignisbenachrichtigungen zu ermöglichen, und zweitens, um die Zahl der Konsumenten skalierbar zu halten.

4.2.4 Sicherheit

Sicherheit gehört wie auch Management zu den klassischen Querschnittstechnologien von Softwaresystemen. Das bedeutet, sie befindet sich nicht ausschließlich in einer bestimmten Schicht, sondern muß vielmehr über alle Schichten hinweg sowohl horizontal als auch vertikal durchgesetzt werden. Zusätzlich müssen oft weitere Maßnahmen außerhalb des Softwaresystems angewendet werden, wie beispielsweise das regelmäßige Erstellen von Sicherheitskopien und Schutz gegen (physikalischen) Diebstahl. Gerade aufgrund dieser Komplexität, die die Erfassung der Sicherheitsproblematik mit sich bringt, und der Tatsache, dass möglicherweise eine einzige Lücke ausreicht, alle anderen Maßnahmen zu umgehen, gilt Sicherheit als eine der am schwierigsten umzusetzenden Anforderung in einem verteilten System.

Neben klassischen Industriebussen wie *PROFIBUS* oder *CAN* werden verstärkt die kostengünstigeren allgegenwärtigen Internettechnologien als Basis für die Gerätekommunikation genutzt, wodurch eine erhöhte Angreifbarkeit gegeben ist. Vor allem vor diesem Hintergrund müssen die Sicherheitsziele der Bewahrung der *Vertraulichkeit*, der Gewährleistung der *Integrität* und der *Verfügbarkeit* der Daten gewährleistet sein. Dazu müssen Frameworks entsprechende Sicherheitsmechanismen mitbringen, die diese Ziele umsetzen [31].

4.2.5 Software-Werkzeuge

Die Akzeptanz von Architekturen und Technologien hängt oftmals davon ab, ob und welche Software-Werkzeuge den Entwicklern und Benutzern zur Verfügung stehen. Das ist umso mehr der Fall, wenn die Zahl der anvisierten Entwickler sehr hoch und nicht auf einige wenige Spezialisten beschränkt ist. Für jede der unterschiedlichen Lebenszyklus-Phasen sollten daher Werkzeuge angeboten werden, die den Beteiligten bei den Aufgaben unterstützen (Abbildung 4.2).

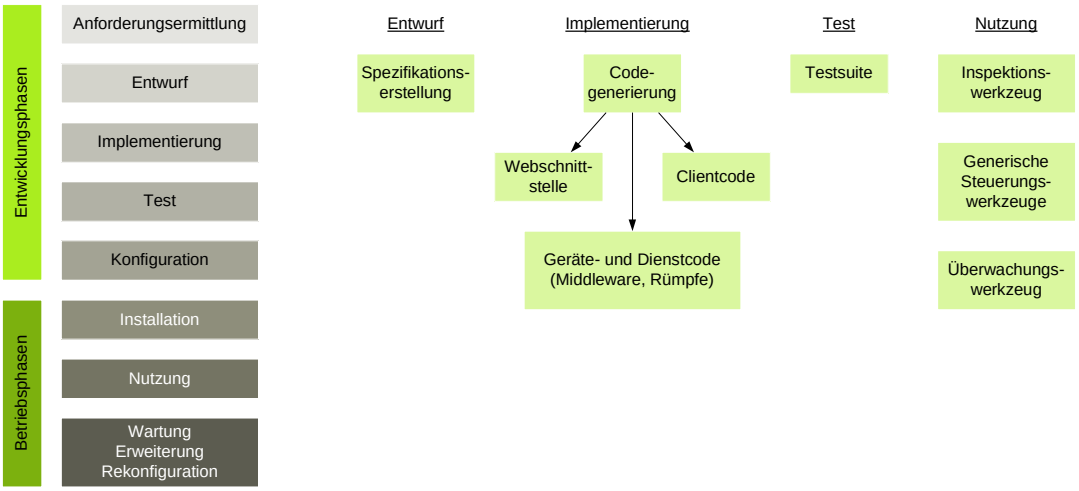

Abbildung 4.2: Unterstützung der Lebenszyklus-Phasen der Dienste durch Software-Werkzeuge

Für die meisten Codegeneratoren bilden formale Beschreibungsdateien die Grundlage zur Erzeugung des Rumpfcodes. Ein Werkzeug sollte den Entwickler in der Entwurfsphase bei der Erstellung der spezifizierenden Dateien helfen. Dadurch können einfache Syntaxfehler vermieden werden bzw. es kann eine sofortige Validierungsprüfung während der Eingabe erfolgen. In einigen Fällen ist es gar nicht möglich, Beschreibungen manuell, beispielsweise durch Eingabe in einen Texteditor, zu erstellen, etwa dann, wenn es sich um ein Binärformat handelt.

In der Entwicklungsphase sollte aus den Beschreibungen mit einem weiteren Werkzeug Code erzeugt werden können. Abhängig von der Zielarchitektur und vom Umfang und Detaillierungsgrad der Spezifikation können dann Programmbausteine für unterschiedlichste Aufgaben generiert werden. Mindestanforderung hier ist der Code, bestehend aus Middleware und Rumpfcode, der auf dem Gerät laufen soll. Weitere Möglichkeiten sind Code für den externen Einsatz, um die Funktionen des Gerätes entfernt zu nutzen, oder zusätzliche Benutzerschnittstellen wie beispielsweise die Generierung einer Webanwendung, über die eine HTML-Seite, die das Gerät repräsentiert, abgerufen werden kann und über die das Gerät ebenfalls gesteuert werden kann.

Kommen wiederkehrende Basisfunktionalitäten zum Einsatz wie etwa Konfigurations- und Verwaltungsdienste, deren Logik weitestgehend statisch ist, so können diese ebenfalls mitgeneriert werden.

Um den Code während der Implementierungsphase auf korrekte Funktionsweise zu überprüfen, sollten entsprechende Testwerkzeuge zur Verfügung gestellt werden. Idealerweise wird spezieller Testcode während der Codegenerierung erzeugt.

Ebenfalls als sinnvoll haben sich generische, also vom verwendeten Gerätetyp unabhängige Steuerungswerkzeuge erwiesen. Diese können sowohl während der Implementierung (zum Testen) als auch später beim Einsatz genutzt werden. Generische Werkzeuge bieten den Vorteil, dass sie nur einmal entwickelt und nur einmal erlernt werden müssen, um anschließend mit ihnen auf unterschiedlichste Geräte oder Komponenten zugreifen zu können (z. B. Webbrowser oder siehe [132] für eine Sprachsteuerung von UPnP-Geräten). Oftmals besitzen sie aber aufgrund ihrer Generik eine umständliche Benutzerschnittstelle oder können nur einen Teil der Funktionsnutzung abdecken.

Mit einem speziellen Werkzeug müssen Geräte während des Betriebes untersucht werden können. Ziel einer solchen Inspektion ist das Auslesen von Informationen über gerätespezifische Daten wie Hersteller, Identifikations- und Versionsnummern, aber auch Angaben über die Funktionen. Der mögliche Informationsumfang hängt direkt vom verwendeten Beschreibungsmodell ab. Ein derartiges Werkzeug ist besonders dann wichtig, wenn das Gerät über keine eigenen physikalischen Benutzerschnittstellen wie Leuchtdioden oder digitale Anzeigen verfügt oder aber wenn es sich aufgrund seines Einsatzes/Einbaus an einer unzugänglichen Stelle befindet.

Während der Nutzungsphase können wichtige Softwareaktualisierungen notwendig sein, beispielsweise dann, wenn eine fehlerhafte Funktionsweise entdeckt wurde und korrigiert werden muss oder aber, wenn Zusatzfunktionen nachgerüstet werden sollen. Um solche Aktualisierungen auch bei laufendem Betrieb zu ermöglichen, sollte es ein Werkzeug geben, mit welchem Geräte vorübergehend gestoppt, aktualisiert und wieder gestartet werden können.

Mit Werkzeugen auf der höheren Anwendungsebene sollten Prozesse definiert werden können, die fähig sind, Geräte in ihre Spezifikation mit einzubeziehen. Die Definition kann dabei textuell bzw. als Skript, besser jedoch grafisch mittels eines Modellierungswerkzeuges erfolgen. Grafische Werkzeuge bilden einen Prozess wesentlich nachvollziehbarer für Menschen ab, wodurch fehlerhafte Definitionen verringert werden. Sowohl die Modellierung der Prozesse als auch deren Ausführung mittels eines Managementsystems sollten Geräte und Softwarekomponenten transparent behandeln.

Software-Werkzeuge können die Akzeptanz einer Technologie erheblich steigern. Die genannten Werkzeuge stellen keine vollständige Aufzählung dar, sie sollten jedoch mindestens in diesem

Umfang zur Verfügung stehen, um den Lebenszyklus der Geräte vollständig zu unterstützen.

4.3 Anforderungen an das Netzwerk und die Kommunikation

4.3.1 Netzwerk

Das Framework muss unabhängig von der verwendeten physikalischen Schicht sein. Das heisst, sowohl drahtloser Netzwerkverkehr als auch das Senden von Daten über Kabel muss möglich sein. Diese Anforderung resultiert direkt aus dem Trend, zunehmend drahtlose Übertragungsmedien wie Bluetooth, WLAN, UMTS oder zukünftig Wireless USB einzusetzen.

Die Unterstützung von IP ist obligatorisch, da es als Netzwerkprotokoll den Standard in der Computervernetzung darstellt und bei seiner Anwendung eine breite Akzeptanz und Interoperabilität zu bestehenden Software- und Hardwaresystemen gewährleistet. Neben der derzeitigen Version IPv4 muss das Framework ebenfalls IPv6 unterstützen, welches einige Beschränkungen von IPv4 aufhebt. IPv6 erweitert die Vorgängerversion um Sicherheits- und Qualitätsmerkmale. Außerdem bringt IPv6 Fähigkeiten zur Autokonfiguration mit, wodurch DHCP theoretisch hinfällig wird. Die wichtigste Neuerung betrifft jedoch die Aufhebung der Adressenbeschränkung von 32 Bit, mit welcher knapp 4,3 Milliarden Adressen dargestellt werden können. Aus historischen Gründen verwalten alleine die USA über 3 Milliarden dieser Adressen, so dass für aufstrebende Länder vor allem im asiatischen Raum wie China, Japan und Korea zu wenig Adressen übrig bleiben. Diese Länder gehören derzeit zu den größten Unterstützern dieser Technologie und setzen aufgrund der Adressknappheit IPv6 bereits aktiv ein.

Im Jahre 2005 wurden weltweit 9 Milliarden Prozessoren produziert[1]. Davon wurden nur 2% für PCs verwendet, die restlichen 98% wurden in eingebettete Geräte verbaut. Betrachtet man diese Zahlen, so wird deutlich, dass zum einen der Großteil der Geräte (fast alle!) auf eingebettete Systeme entfällt und zum anderen die Gesamtzahl der in nur einem Jahr produzierten Prozessoren bereits die Zahl der zur Verfügung stehenden IPv4-Adressen um mehr als das zweifache übersteigt.

4.3.2 Kommunikation

In geräteintegrierenden Frameworks und in verteilten Softwaresystemen allgemein müssen Komponenten bekanntgemacht, gesteuert, überwacht und verwaltet werden können. Dazu benötigen sie eine Kommunikationsinfrastruktur, die für die jeweiligen Aufgaben geeignete Kommunikationsmuster anbietet.

[1] `http://www.netrino.com/Publications/Glossary/index.php`

Die Kommunikation aller Komponenten sollte aus den gleichen Gründen wie beim Plug-and-Play unabhängig von anderen (zentralen) Komponenten erfolgen, also auf Peer-to-Peer-Basis stattfinden können. Sie sollte nicht eine übergeordnete Komponente erforderlich machen, die die Kommunikation verwaltet und koordiniert.

Für ein effizientes Plug-and-Play sowie für Ereignisnachrichten sollte ein Mechanismus bereitgestellt werden, der es einem Gerät erlaubt, eine Nachricht gleichzeitig an mehrere Partner zu senden, ohne zuvor individuelle Peer-to-Peer-Verbindungen mit jedem Partner zu erstellen. In den meisten Frameworks wird dieses Muster mit einem Gruppenkonzept realisiert. Nachrichten werden an eine Gruppe geschickt und Komponenten, die dieser Gruppe angehören, erhalten die Nachricht.

Weiterhin müssen die Kommunikationsmuster *Anfrage-Antwort* (*Request-response*) und *Aufruf* (*Call*) unterstützt werden. Anfrage-Antwort wird immer dann eingesetzt, wenn die aufgerufene Komponente Daten in der Antwort zurückliefern soll bzw. dann, wenn der Sender eine Bestätigung für den Empfang der Anfrage erhalten möchte. Aufruf wird dagegen eingesetzt, wenn der Empfänger keine Antwortdaten zurücksendet und wenn der Sender keine Bestätigung für den Empfang des Aufrufes benötigt.

In verteilten Systemen, in denen weitestgehend voneinander unabhängige Komponenten miteinander kommunizieren, kann nicht immer sichergestellt werden, ob und wie schnell eine Komponente auf eine Anfrage antwortet. Daneben gibt es Prozesse, die in der Natur der Sache liegend eine sehr lange Bearbeitungsdauer aufweisen, etwa bei der Berechnung großer Datenmengen unter Verwendung aufwändiger Algorithmen (z. B. Video-Encoding). Damit die aufrufende Entität nicht dazu gezwungen wird, ihre Arbeit zu unterbrechen und zu blockieren bis die Antwort eintrifft, sollte die Kommunikationsinfrastruktur asynchronen Nachrichtenaustausch ermöglichen. Dadurch kann eine wesentlich höhere Flexibilität und Skalierbarkeit gewährleistet werden. Außerdem können Ressourcen geschont werden, da beispielweise Netzwerkverbindungen nicht offengehalten werden müssen.

4.4 Fazit

Das Kapitel führte allgemeine Anforderungen an universell einsetzbare Frameworks auf. Gegenwärtig existiert kein Framework, welches diese Anforderungen vollständig erfüllt bzw. eine dominante Stellung vorweisen kann. Insbesondere domänenspezifische Technologien wie industrielle Bussysteme verhindern einen breiten Einsatz.

Das Devices Profile for Web Services gilt als ein vielversprechender Ansatz, die bestehenden Probleme zu lösen. Zusammen mit den anderen Web-Services-Spezifikationen und der Nähe zu

den serviceorientierten Architekturen besitzt es derzeit als einziges das Potenzial Domänen zu überwinden und tatsächlich Geräte in eine serviceorientierte IT-Landschaft zu integrieren.

Neben DPWS wird in dieser Arbeit auch Universal Plug and Play (Kapitel 6) bei der Gegenüberstellung mit der Web-oriented Device Architecture betrachtet, da UPnP ähnlich universelle Ziele wie DPWS verfolgt.

5 Umsetzung einer serviceorientierten Architektur mit dem Devices Profile for Web Services

Serviceorientierte Architekturen stellen derzeit ein bedeutendes Paradigma dar, um die zunehmende Softwarekomplexität bei gleichzeitig höher werdenden Anforderungen an Flexibilität und Wiederverwendbarkeit in den Griff zu bekommen. Während SOA jedoch in erster Linie ein konzeptionelles und organisatorisches Paradigma ist, werden auf der anderen Seite geeignete (universell anwendbare) Technologien benötigt, um eine SOA umzusetzen.

Das Devices Profile for Web Services (DPWS) liefert eine solche konkrete Technologie. DPWS spezifiziert ein Profil, welches mehrere Web-Services-Protokolle verwendet. Die Protokolle sowie das Profil wurden im Abschnitt 2.3 und im Abschnitt 2.8 vorgestellt.

Um DPWS praktisch evaluieren zu können, wird eine Software benötigt, die diese Spezifikation umsetzt. Zum Zeitpunkt der Arbeit existierte lediglich eine inoffizielle, prototypische, noch auf einer älteren DPWS-Version basierende Lösung. Diese wurde innerhalb des SIRENA-Forschungsprojektes erarbeitet [87].

Aus dieser Situation heraus wurde die *Web Services for Devices* (WS4D) Initiative gegründet, um mehrere interoperable DPWS-Stacks zu schaffen [97]. Innerhalb von WS4D werden drei Softwarelösungen entwickelt, die unterschiedliche Anwendungsdomänen adressieren und sich gegenseitig ergänzen:

- WS4D-gSOAP ist in der Programmiersprache C realisiert und setzt auf gSOAP [133] auf, welches bereits Implementierungen für wichtige Web-Services-Protokolle enthält. WS4D-gSOAP ist vor allem für ressourcenarme, eingebettete Systeme geeignet.
- WS4D-JavaME basiert auf das *Connected Limited Device Configuration* (CDC/CLDC) Profil und ist daher für den Einsatz auf eingebetteten Geräten (PDAs, Mobiltelefone usw.) konzipiert, auf welchen sich eine JavaME (Java Micro Edition) Virtual Machine befindet.
- WS4D-Axis2 ist eine auf JavaSE (Java Standard Edition) basierende DPWS-Implementierung. Sie setzt auf den SOAP-Prozessor Axis2 von Apache auf. WS4D-Axis2 ist vor allem für Dienste auf höherer Ebene und für Clientimplementierungen geeignet, die DPWS-fähige Geräte verwenden wollen.

Im nachfolgenden Kapitel wird die auf Java basierende DPWS-Implementierung für das Apa-

che-Axis2-Projekt vorgestellt, die durch den Autor dieser Arbeit realisiert wurde. Anschließend wird gezeigt, wie eine serviceorientierte Architektur mit DPWS umgesetzt werden kann. Dazu wird exemplarisch eine Lokalisierungsplattform als Anwendungsdomäne aufgebaut. Bei der prototypischen Umsetzung kamen der WS4D-gSOAP- und der WS4D-Axis2-Stack zum Einsatz.

Weiterhin wird ein eigener Ansatz für eine formale Definition von Geräte- und Diensttypen für DPWS beschrieben. Eine solche Typisierung existiert bisher nicht, sie ist jedoch für Gerätehersteller und Entwickler aus Interoperabilitätsgründen sehr wichtig.

5.1 DPWS-Implementierung für Apache Axis2

Für die praktische Evaluierung von DPWS wird eine geeignete Middleware benötigt. Dazu wurde die Spezifikation des Devices Profile for Web Services für den auf Java basierenden SOAP-Prozessor Axis2 [134] prototypisch implementiert. Es stehen sowohl Programmbibliotheken für Client-Anwendungen als auch für die Umsetzung von Geräten/Diensten zur Verfügung.

5.1.1 Apache Axis2

Apache Axis2 ist ein modular aufgebauter SOAP-Prozessor, welcher SOAP-Nachrichten empfangen, verarbeiten und versenden kann. Das alles geschieht für den Entwickler größtenteils transparent, so dass sich dieser hauptsächlich um die Geschäftslogik kümmern kann, ohne etwas von den darunterliegenden Mechanismen wie (De-)Serialisierung von Datenklassen, Aufrufen der richtigen Methoden in den Dienstklassen, Verwendung von synchronen oder asynchronen Kommunikationsmustern usw. mitzubekommen.

Axis2 wird vor allem auf Enterprise-Servern eingesetzt. Als Nachfolger des weit verbreiteten Axis-Projektes wurde er komplett überarbeitet und gilt zurzeit als eine der fortschrittlichsten und funktionsreichsten SOAP-Bibliotheken. Axis2 ist ein Open-Source-Projekt und wird unter der Apache-Organisation [135] weiterentwickelt. Die wichtigsten Funktionalitäten und Charakteristiken (die für die Realisierung von DPWS relevanten sind hervorgehoben) sind:

- *Unabhängigkeit von der verwendeten Transportschicht*, enthalten sind HTTP, SMTP, TCP und JMS
- Unterstützung der Protokolle SOAP 1.1, *SOAP 1.2*, *WSDL 1.1*, WSDL 2.0, MTOM, XOP, SwA, *WS-Addressing*, WS-Policy
- Unabhängigkeit vom verwendeten Data Binding, ADB und XMLBeans sind vorhanden
- Hinzufügen und Entfernen von Diensten zur Laufzeit (*Hot Deployment*)
- *Synchrone und asynchrone Aufrufe* sowohl auf Anwender- (mittels Callbacks) als auch auf Transportschicht (durch Verwendung von WS-Addressing)

- *Erweiterbare modulare Architektur*
- *WSDL-zu-Java-* und Java-zu-WSDL-*Codegenerator*, auch als Eclipse-Plug-in
- *Installation* sowohl als Webarchiv in einem J2EE-konformen Servletcontainer (z. B. Tomcat) als auch eigenständig *als allein operierendes Programm* (Standalone)

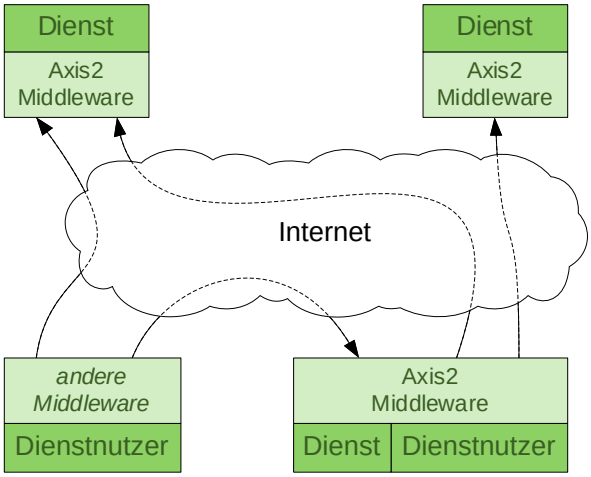

Abbildung 5.1: Gleichzeitige Verwendung von Axis2 als Dienst und Dienstnutzer

Die Web-Services-Protokollfamilie besteht aus einer Menge kombinierbarer, lose miteinander verbundener Protokolle, die mit dem Ziel entworfen wurden, aus ihnen neue Spezifikationen (z. B. Profile) für konkrete Anwendungsgebiete zu erstellen. Die Architektur von Axis2 spiegelt genau diese Vorgehensweise wider. Sie ist so ausgelegt, dass zusätzliche (oder zukünftige) Protokolle als Modul implementiert und nach Bedarf installiert werden können, ohne dass vorhandene Module (Protokolle) adaptiert werden müssen.

Die Axis2-Middleware kann für Dienste und von Dienstnutzern verwendet werden, auch gleichzeitig, wie in Abbildung 5.1 dargestellt. Auf der Dienstseite hat Axis2 die Aufgabe SOAP-Nachrichten zu empfangen, die SOAP-Header zu verarbeiten und die Nachricht an die richtige Anwendung (Dienst) als transparenten Methodenaufruf weiterzuleiten. Eingehende SOAP-Nachrichten werden von *TransportListener* empfangen. Für jedes Transportprotokoll, welches unterstützt werden soll, muss mindestens ein entsprechender Listener installiert sein. Dieser erstellt aus der Nachricht einen *SOAPEnvelope* und reicht diesen an die *eingehende Prozesskette* weiter. Eine Prozesskette besteht aus sequenziell angeordneten *Phasen* und diese wiederum aus *Handler* (Abbildung 5.2). Jeder Handler inspiziert nacheinander die SOAP-Header und kann entsprechende Parameter für die weitere Verarbeitung setzen. Beispielsweise werden innerhalb

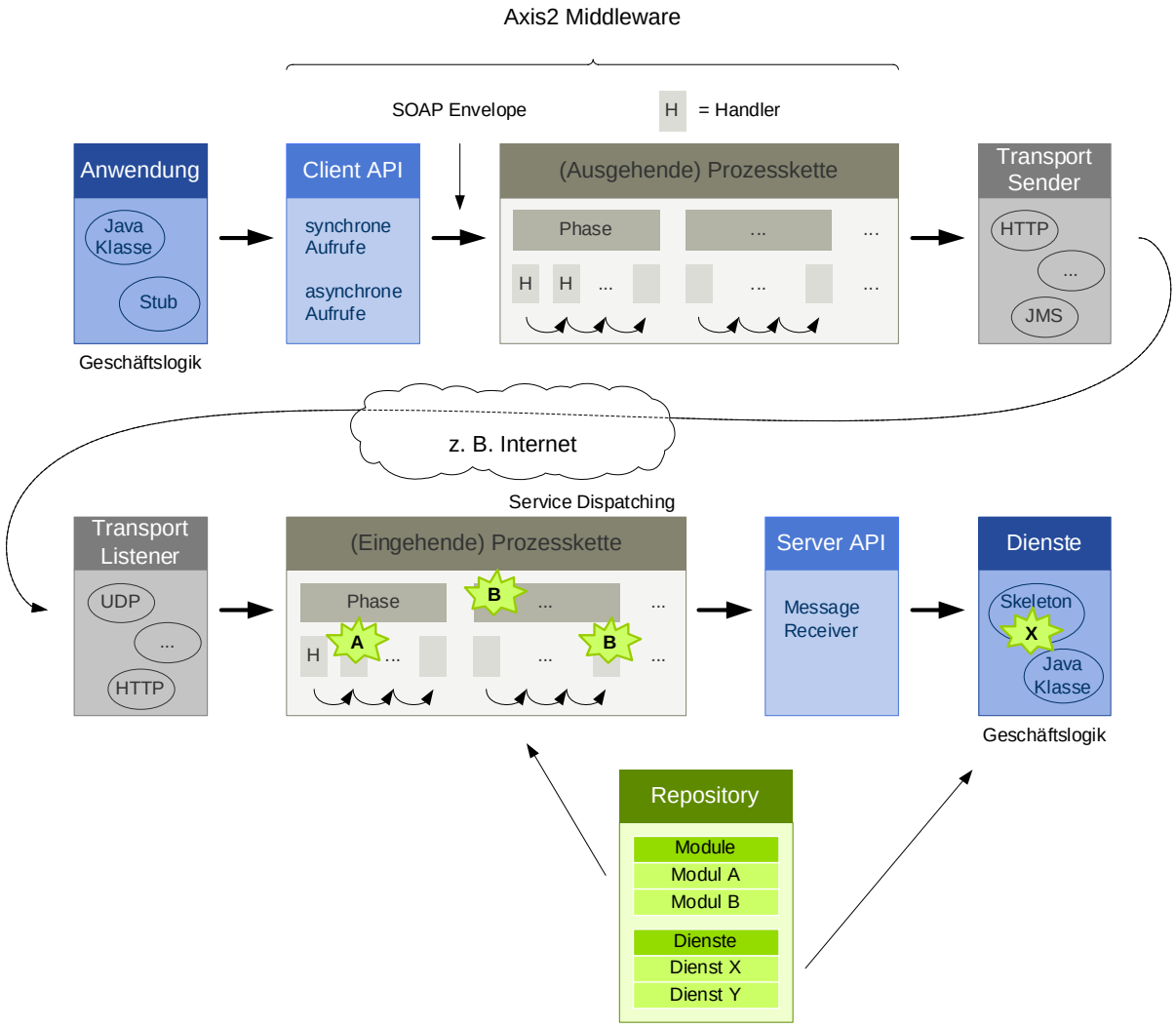

Abbildung 5.2: Fluss einer SOAP-Nachricht innerhalb der Axis2-Architektur

der eingehenden Prozesskette WS-Addressing-Informationen ausgewertet, falls diese vorhanden sind. Jede Phase steht für eine bestimmte Aufgabe. So müssen die Handler während der *Dispatch*-Phase den Dienst und die Operation ermitteln, für den die Nachricht bestimmt ist. Nach Abschluss der Prozesskette wird der SOAP-Body in ein Java-Objekt konvertiert (*Marshalling*) und die Dienstklasse mit der entsprechenden Methode aufgerufen.

Auf der Dienstnutzerseite arbeitet Axis2 ähnlich. Die Anwendung, die den entfernten Dienst nutzen möchte, ruft eine Methode (die für eine bestimmte WSDL-Operation steht) in einem Proxy-Objekt auf. Axis2 konvertiert die übergebenen Parameter in XML-Strukturen (*Demarshalling*) und startet die *ausgehende Prozesskette*. Auch hier werden nacheinander Handler in unterschiedlichen Phasen aufgerufen, mit dem Unterschied, dass nun SOAP-Header der Nachricht hinzugefügt werden. Beispielsweise könnten WS-Security-Handler die Nachricht signieren und den Schlüssel als SOAP-Header ablegen. Nach Abschluss der Prozesskette verschickt ein *TransportSender* die Nachricht über das gewählte Medium.

Die Reihenfolge der Phasen und die der Handler können in der globalen Konfigurationsdatei *axis2.xml* eingestellt werden. Diese Datei enthält auch alle anderen Einstellungen wie Netzwerkparameter, zu verwendende Module, Clustereinstellungen und *Observer*, die den Installationsprozess von Diensten überwachen. Module können zusätzliche Handler in die Prozessketten und zusätzliche Dienste installieren, siehe Abbildung 5.2. Dadurch kann das Gesamtverhalten des Systems geändert werden. Alle Axis2-Erweiterungen für DPWS wie Discovery und Eventing wurden als Module realisiert.

Module und Dienste werden in einem *Repository* organisiert, welches Axis2 beim Starten zusammen mit der Konfigurationsdatei übergeben werden muss. Sowohl Module als auch Dienste sind gepackte Archive und müssen eine bestimmte Struktur aufweisen.

5.1.2 UDP als Transportprotokoll für SOAP

Axis2 ist eine SOAP-Engine, die unabhängig vom Transportprotokoll entworfen wurde. Das heißt, eine SOAP-Nachricht kann beispielsweise in transparenter Weise HTTP empfangen und über Mail weitergeschickt oder beantwortet werden. Im Umfang von Axis2 sind Transportmodule für TCP, Mail, HTTP und JMS enthalten. Ein entsprechendes Modul für UDP gibt es nicht. Da SOAP-over-UDP jedoch Voraussetzung für die Discovery-Implementierung ist, wurde ein Plug-in implementiert. Im Folgenden wird beschrieben, welche Eingriffe dazu notwendig waren und wie das Plug-in angewendet werden kann.

Das *soapudp*-Modul stellt dazu die zwei Klassen *UDPTransportSender* zum Senden und *UDPTransportReceiver* zum Empfangen von Datagrammen bereit. Die Instanzen werden, wie bei Axis2 üblich, in der globalen Konfigurationsdatei *axis2.xml* erzeugt und konfiguriert. Außer-

dem muss das Modul im Repository liegen und in der Konfigurationsdatei referenziert werden (Listing 5.1).

Listing 5.1: Konfiguration des *soapudp*-Moduls in der axis2.xml

```
<axisconfig name="AxisJava2.0">
    ...
    <module ref="soapudp" />
    <transportReceiver name="wsdmc"
        class="org.ws4d.axis2.soapudp.UDPTransportReceiver">
        <parameter name="port">3702</parameter>
        <parameter name="group">239.255.255.250</parameter>
        <parameter name="host">139.30.201.170</parameter>
    </transportReceiver>
    <transportReceiver name="soapudp"
        class="org.ws4d.axis2.soapudp.UDPTransportReceiver">
        <parameter name="port">7500</parameter>
        <parameter name="host">139.30.201.170</parameter>
    </transportReceiver>
    <transportSender name="soapudp"
        class="org.ws4d.axis2.soapudp.UDPTransportSender">
    </transportSender>
    ...
</axisconfig>
```

Es können beliebig viele UDP-Sender und -Empfänger angelegt werden. Sie können später im Quelltext über ihre Namen identifiziert werden. Sender und Empfänger mit gleichlautenden Namen werden jeweils an den gleichen Socket gebunden. Die Konfiguration der Empfänger und Sender ist sehr flexibel gehalten. Für einen empfangenden Socket muss mindestens der *port* angegeben werden. Der *host* wird nur dann benötigt, wenn Datagramme aus einem bestimmten Subnetz empfangen oder in dieses versendet werden sollen. Diese Konfigurationsmöglichkeit ist wichtig, um beispielsweise einen Discovery-Proxy zu realisieren, da dieser eine feinere Kontrolle über die empfangenen und zu sendenden Nachrichten benötigt. Wird der *group*-Parameter spezifiziert, wird der Socket an die angegebene Multicastgruppe gebunden. Bei einem UDP-Sender kann nur ein *host* angegeben werden, der Port wird zur Laufzeit dynamisch vom Betriebssystem ausgewählt, es sei denn, der Name des Senders stimmt mit einem Empfänger überein. In diesem Fall erbt der Sender diesen Parameter. Auf diese Weise kann *ein* Socket gleichzeitig zum Senden und Empfangen verwendet werden (wichtige Voraussetzung für die Discovery-Umsetzung). Da ein Rechner in der Regel über mehr als eine IP-Adresse verfügt (mindestens noch die des lokalen Hosts), sollte *host* immer angegeben werden, da sich das Betriebssystem ansonsten eine der vorhandenen IP-Adressen heraussucht.

Der *UDPTransportReceiver* wird eingesetzt, um UDP-Nachrichten an einem lokalen Socket zu empfangen. Die Nachricht wird geparst, als SOAP-Nachricht aufbereitet und an die *AxisEn-*

gine weitergereicht. Nachdem eine Nachricht empfangen und erfolgreich geparst wurde, werden folgende Eigenschaften im *MessageContext* gesetzt, die von den nachfolgenden Handlern ausgewertet werden können:

`UDPTransportReceiver.UDP_SOURCE_ADDRESS`	Adresse des Absenders
`UDPTransportReceiver.UDP_SOURCE_PORT`	Port des Absenders
`UDPTransportReceiver.UDP_RECEIVER_ADDRESS`	Empfangsadresse
`UDPTransportReceiver.UDP_RECEIVER_PORT`	Empfangsport
`UDPTransportReceiver.UDP_RECEIVER_GROUP`	Empfangsgruppe

Als eines der nächsten Handler – die tatsächliche Reihenfolge hängt von den installierten Modulen ab – wird der *UDPMessageFilter* aufgerufen. Dieser filtert anhand der *MessageID* bereits einmal empfangene Nachrichten und unterbindet die weitere Verarbeitung durch Axis2. Beispielsweise werden laut den vorgeschlagenen Werten von SOAP-over-UDP Multicast-Nachrichten viermal verschickt. Der *UDPMessageFilter* sorgt dafür, dass die Nachricht trotzdem nur einmal an die eigentliche Anwendung (Dienst) übergeben wird.

Der *UDPTransportSender* wird verwendet, wenn eine SOAP-Nachricht über UDP verschickt werden soll. Die Zieladresse und der Zielport werden zur Laufzeit angegeben. Dazu gibt es zwei Möglichkeiten. Der *UDPTransportSender* verwendet folgende Reihenfolge, um das Ziel zu ermitteln: Das URI-Schema wird ermittelt, indem die Adresse der Endpoint-Referenz des *wsa:To*-Headers aus dem *MessageContext* extrahiert wird. Beginnt diese mit *soap.udp://*, wird diese Adresse verwendet. Trifft ersteres nicht zu, müssen im *MessageContext* die folgenden Eigenschaften gesetzt sein:

`UDPTransportSender.UDP_DESTINATION_ADDRESS`	Zieladresse
`UDPTransportReceiver.UDP_DESTINATION_PORT`	Zielport

Der *UDPTransportSender* kann auch angewiesen werden, die gleiche SOAP-Nachricht mehrmals zu verschicken. Das *soapudp*-Modul implementiert dazu den Retransmission-Algorithmus von SOAP-over-UDP (Abs. 2.3.1). Der Transport-Sender kümmert sich eigenständig um die Einhaltung der (zufälligen) Wartezeiten und garantiert, dass die Nachricht wiederholt und unverändert gesendet wird. Um *UDPTransportSender* anzuweisen, eine Nachricht mehrmals zu verschicken, muss eine Instanz der Klasse *UDPRetransmissioner* im *MessageContext* hinterlegt sein[1]. Im *UDPRetransmissioner* können die Anzahl der Wiederholungen und die Zeiten zwischen den Sendungen sowohl für Unicast- als auch für Multicast-Zieladressen eingestellt werden. Die

[1] Eigenschaft `UDPRetransmissioner.PARAM_UDP_RETRANSMISSIONER_INSTANCE`

voreingestellten Werte entsprechen denen aus *Appendix I* der SOAP-over-UDP-Spezifikation.

Das *soapudp*-Modul wurde so entworfen, dass es unabhängig von anderen Erweiterungen in Axis2 eingesetzt werden kann. Es unterstützt folgende Funktionalitäten:

- Empfangen von UDP-Nachrichten an einer lokalen Adresse (Unicast oder Multicast)
- Senden von SOAP-Nachrichten über UDP
- Optionale Verwendung des Retransmission-Algorithmus aus SOAP-over-UDP zum mehrfachen Senden der gleichen SOAP-Nachricht
- Filtern von empfangenen Nachrichten mit gleichen IDs
- Gleichzeitige Verwendung eines Sockets zum Senden und Empfangen

5.1.3 Die Discovery-Erweiterung

Das WS-Discovery-Protokoll ermöglicht das dynamische Entdecken bzw. die Bekanntgabe von Diensten (*Target Services*) zur Laufzeit und stellt damit einen Mechanismus für Ad-hoc-Verhalten von Web Services zur Verfügung. Ohne diesen Mechanismus werden Web Services unter Axis2 einfach installiert und stehen dann an der Installationsadresse zur Verfügung. Potenzielle Dienstnutzer werden dessen jedoch nicht gewahr. Sie müssen nicht nur die Adresse kennen, sondern sie müssen auch davon ausgehen, dass der Dienst tatsächlich zur Verfügung steht.

Die Discovery-Erweiterung für Axis2 bietet Funktionalitäten für beide Seiten: Die Dienstnutzer können mit der *DiscoveryClientAPI* nach Diensten suchen und sie zur Laufzeit binden, und Dienste können sich mit der *DiscoveryTSAPI* am Netzwerk an- und abmelden sowie auf Anfragen von Dienstnutzern antworten.

Die Erweiterung besteht aus einem Discovery-Modul und einem Discovery-Service. Beide müssen sowohl auf der Dienstnutzer- als auch auf der Dienstseite installiert werden. Das Discovery-Modul installiert den *Discovery-Service* sowie die beiden Handler *DiscoveryDispatcher* und *DiscoveryOutHandler* in die Prozesskette der Axis2-Engine. Der *DiscoveryDispatcher* fängt Discovery-Nachrichten ab und leitet sie an den *Discovery-Service* weiter, wo sie ausgewertet und beantwortet werden. Dieser Vorgang wird von den installierten Diensten nicht bemerkt, kann jedoch durch das Setzen eines anwendungsspezifischen *Delegates*[2] beeinflusst werden. Diese Möglichkeit ist beispielsweise für die Umsetzung eines Discovery-Proxies wichtig. Dieser muss ein Verhalten implementieren, welches vom Standard-Verhalten abweicht. Der *DiscoveryOutHandler* setzt die *Application Sequence* der Nachrichten als SOAP-Header, damit die UDP-Datagramme gegebenenfalls auf der Empfängerseite in die richtige Reihenfolge gebracht werden können (Abs.

[2] Mittels einer Delegation kann eine Klasse die Ausführung an eine andere abgeben (delegieren). Delegation wird oft als Alternative zur objektorientierten Vererbung verwendet, da sie in der Praxis zu weniger statischen Abhängigkeiten führt.

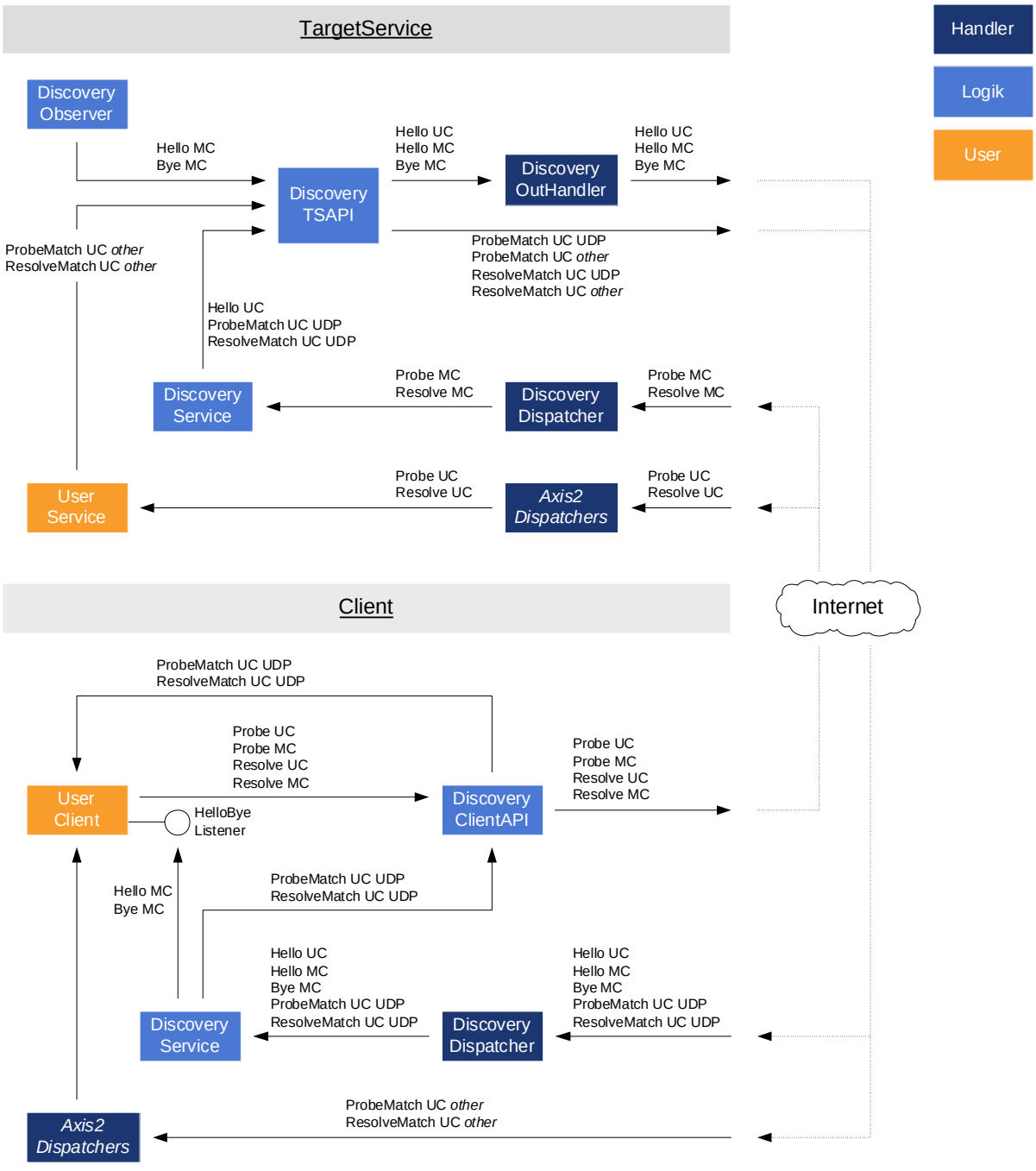

Abbildung 5.3: Nachrichtenfluss innerhalb des Discovery-Moduls

2.3.6).

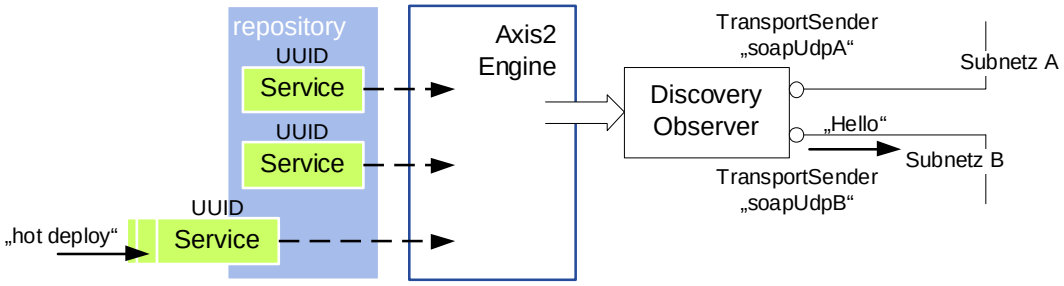

Abbildung 5.4: Funktionsweise des *Discovery Observers*

Ein Ziel bei der Entwicklung der Discovery-Erweiterung war es, Dienst-Entwickler weitestgehend von den internen Discovery-Mechanismen zu befreien. Im Idealfall sollte ein Dienst einmal konfiguriert werden und danach alle Discovery-Nachrichten automatisch gesendet und Anfragen automatisch beantwortet werden, so dass sich die Discovery-Schicht ähnlich zur Netzwerkschicht transparent verhält. Axis2 bietet die Möglichkeit, Überwacher (*Observer*) global zu konfigurieren, die benachrichtigt werden, wenn Dienste installiert, gestartet, gestoppt und wieder deinstalliert werden. Für die Discovery-Erweiterung wird genau dieser Mechanismus benötigt und durch den *DiscoveryObserver* umgesetzt (Abbildung 5.4).

Listing 5.2: Konfiguration eines *Target Services* in der services.xml

```
<serviceGroup>
    <service name="BlueScanServerService">
        ...
        <parameter name="Discovery" auto="true">
            <enableAutoDiscovery>true</enableAutoDiscovery>
            <UUID>urn:uuid:A3CE4B9E4585FF5BDF1183465330846</UUID>
            <types>
                <type ns="http://schemas.xmlsoap.org/ws/2006/02/devprof"
                    prefix="wsdp" name="Device" />
                <type ns="http://www.ws4d.org/bluescan"
                    prefix="bs" name="BlueScanServerService" />
            </types>
            <scopes>
                <scope uri=
                    "http://schemas.xmlsoap.org/ws/2005/04/discovery/adhoc" />
                <scope uri="socom:non-public-area" />
            </scopes>
            <UDPTransportSenders>
                <sender name="soapudpB" />
            </UDPTransportSenders>
        </parameter>
    </service>
</serviceGroup>
```

Nach einer Installation oder Deinstallation eines Service-Archivs schickt der Axis2-Prozessor ein entsprechendes Ereignis an den *DiscoveryObserver*, der daraufhin die Discovery-relevanten Daten aus der *services.xml* (Listing 5.2) liest und je nach Konfiguration den Dienst im Netz bekanntgibt oder abmeldet. Die Konfigurationsdatei muss eine eindeutige, unveränderbare UUID für den Dienst spezifizieren. Diese wird vom *DiscoveryObserver* für die *Hello*- und *Bye*-Nachrichten verwendet, und stellt sicher, dass der Dienst im Falle einer Netzwerkänderung (z. B. andere IP-Adresse) jederzeit als derselbe Dienst identifiziert werden kann. Weiterhin kann hier eine Restriktion auf Transportebene vorgenommen werden, indem nur die *TransportSender* angegeben werden, über die die Nachrichten geschickt werden sollen (dürfen). Im Beispiel in der Abbildung 5.4 werden die Nachrichten nur in das Subnetz B gesendet, siehe Listing 5.2. Diese Vorgehensweise war notwendig, um einen Discovery Proxy umzusetzen.

Die Discovery-Erweiterung bietet Dienstnutzern die Möglichkeit an, das Netzwerk auf An- und Abmelden von Diensten zu überwachen sowie aktiv nach ihnen zu suchen. Dafür steht die *DiscoveryClientAPI*-Bibliothek mit statischen Methoden zur Verfügung. Das Überwachen des An- und Abmeldens der Dienste wurde, wie in Java üblich, mit dem *Listener*-Konzept realisiert. Interessenten können sich an der Client-API als *HelloByeListener* registrieren und erhalten in der Folge die Dienste als *ProxyTargetService*-Objekte abgebildet. Um die logischen Adressen (UUID) der Dienste in physikalische aufzulösen, kann in den Objekten die *resolve*-Methode aufgerufen werden. Diese Funktionalitäten, wie auch das Senden von Suchanfragen (*Probe*-Nachrichten), geschieht für den Dienstnutzer transparent. Bei der Suche nach Diensten kann der Dienstnutzer verschiedene Optionen angeben, wie zu unterstützende Diensttypen, den Scope und das Subnetz. Für Netzwerke mit einer hohen Anzahl an potenziell passenden Diensten kann auch eine zulässige Höchstzahl an Diensten angegeben werden, die in der Antwort enthalten sein dürfen.

5.1.4 Die Eventing-Erweiterung

WS-Eventing stellt einen standardisierten Mechanismus für Web Services zur Verfügung, der Diensten das Anbieten und Versenden von ereignisbasierten Nachrichten als Teil ihrer Schnittstelle erlaubt und Dienstnutzer befähigt, sich für diese Nachrichten zu registrieren und sie zu empfangen.

Die Eventing-Erweiterung für Axis2 bietet wiederum Funktionalitäten für beide Seiten: Dienstnutzer können mit der *EventingClientAPI* Ereignisnachrichten abonnieren, die Gültigkeit der Abonnements verlängern und sie wieder aufheben. Dienste können den *SubscriptionManagerService* installieren und verwenden, um Abonnementanfragen zu empfangen, die Gültigkeitsdauer einzustellen und die Abonnements zu verwalten.

Die zentrale Komponente auf der Produzentenseite ist die *SubscriptionManagerAPI*. Diese

verwaltet für jedes Abonnement einen *SubscriberProxy*, der den entfernten *Subscriber* repräsentiert. Der *SubscriberProxy* speichert Gültigkeitsdauer, die Filtereinstellungen und den Endpunkt des Konsumenten, an den die Ereignisnachrichten geschickt werden sollen.

Den möglichen Ablauf in der vorgegebenen Implementierung zeigt das Sequenzdiagramm (Abbildung 5.5). Erreicht eine *Subscribe*-Anfrage einen Dienst, leitet dieser die Anfrage an die *SubscriptionManagerAPI* weiter, welche die Nachricht parst. Ist die Anfrage erfolgreich, legt die *SubscriptionManagerAPI* einen neuen *SubscriberProxy* an und sendet eine entsprechende Antwort an den Subscriber zurück. Ob die Anfrage erfolgreich ist, hängt von verschiedenen Parametern ab. In der vorgegebenen Implementierung der *SubscriptionManagerAPI* muss beispielsweise der Benachrichtigungsmodus auf *Push* gesetzt sein. Die Gültigkeitsdauer wird auf einen festen Wert gesetzt, welcher in der Konfigurationsdatei des jeweiligen Dienstes eingestellt werden kann. Tritt nun ein Ereignis auf, kann sich der Dienst die gültigen Abonnenten aus der *SubscriptionManagerAPI* holen. Für jeden *SubscriberProxy* wendet der Dienst den entsprechenden Filter an, um zu ermitteln, ob die Ereignisnachricht verschickt werden soll. Ist dieses der Fall, sendet er die Ereignisnachricht an die angegebene *NotifyTo*-Adresse des *SubscriberProxies*.

Eintreffende *Renew*- und *Unsubscribe*-Nachrichten gehen zunächst direkt an den *SubscriptionManagerService*, da dieser in der *SubscribeResponse* als *SubscriptionManager* referenziert wurde. Der *SubscriptionManagerService* leitet die Nachrichten an die *SubscriptionManagerAPI* weiter, die daraufhin die *SubscriberProxies* aktualisiert bzw. entfernt.

Die *SubscriptionManagerAPI* übernimmt eine weitere, hier nicht dargestellte Aufgabe. Sie überprüft in regelmäßigen Abständen, ob die *SubscriberProxies* noch gültig sind. Ist dieses nicht mehr der Fall, werden sie verworfen. Damit ist sichergestellt, dass sich nach jeder Überprüfung ausschließlich aktuelle Abonnements auf der Dienstseite befinden.

Die vorgegebene Implementierung (Abbildung 5.6) kann auch durch eine benutzerspezifische ersetzt werden. Dazu leitet der Dienst eine *Subscribe*-Anfrage nicht an die *SubscriptionManagerAPI* weiter, sondern bearbeitet sie selbst und gibt seinen eigenen Endpunkt als *SubscriptionManagerService* an. In diesem Fall erreichen alle weiteren Eventing-Nachrichten nicht den installierten *SubscriptionManagerService*, sondern den Dienst selbst.

Das Eventing-Modul installiert auf der Produzentenseite zwei Handler in die Prozesskette. Der *EventingInHandler* und *EventingOutHandler* lesen bzw. setzen den *wse:Identifier*-Header (Abs. 2.3.7), über den der *SubscriptionManagerService* den entsprechenden *SubscriberProxy* identifizieren kann, für den die Anfrage gilt (Tatsächlich werden die *SubscriberProxies* in der Implementierung über diese IDs verwaltet).

Für Dienstnutzer steht die *EventingClientAPI* zur Verfügung. Sie enthält im Wesentlichen statische Methoden, um Ereignisse zu abonnieren, die Abonnements zu verlängern und zu be-

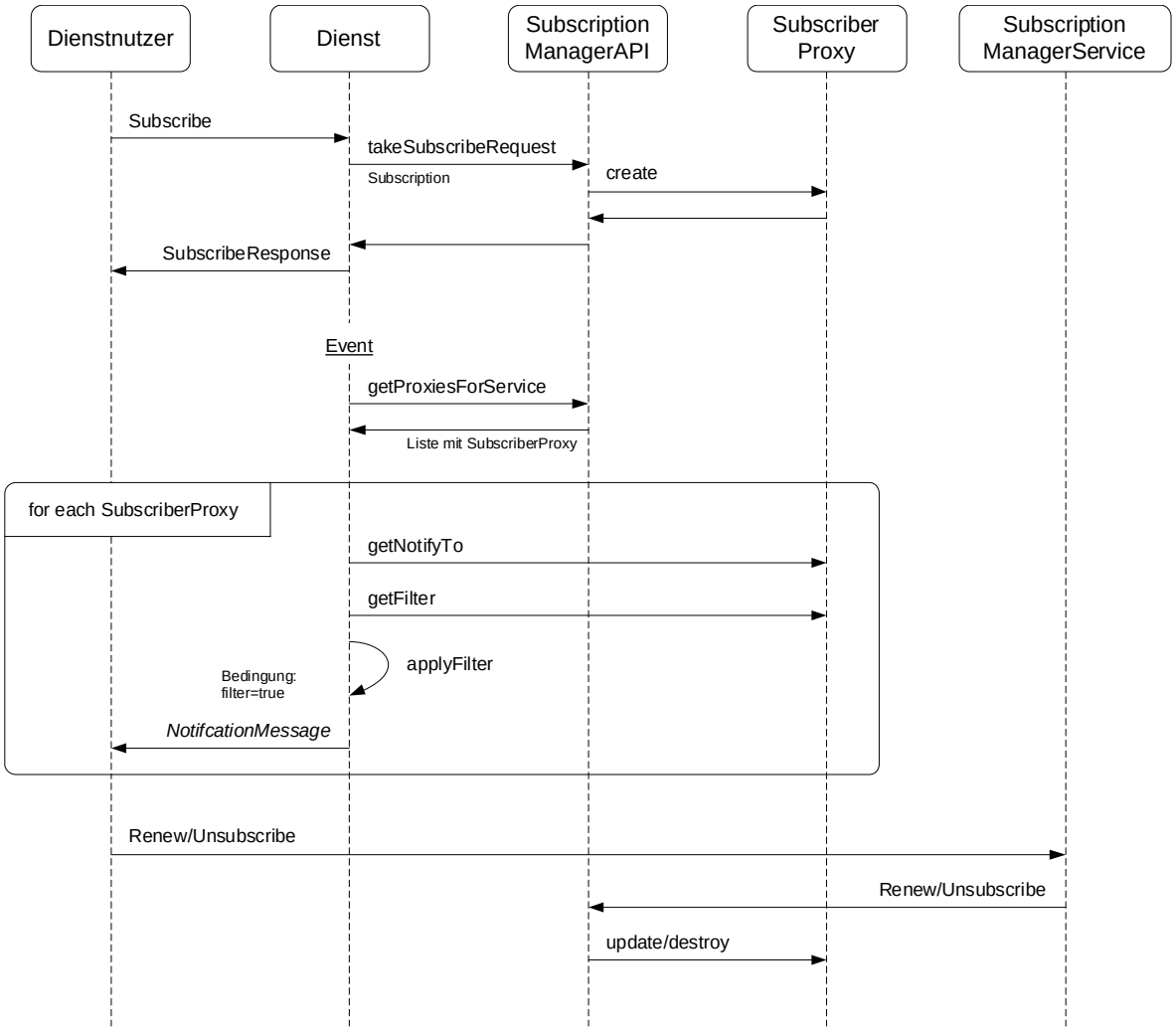

Abbildung 5.5: Sequenzdiagramm für Arbeitsweise des Eventing-Moduls

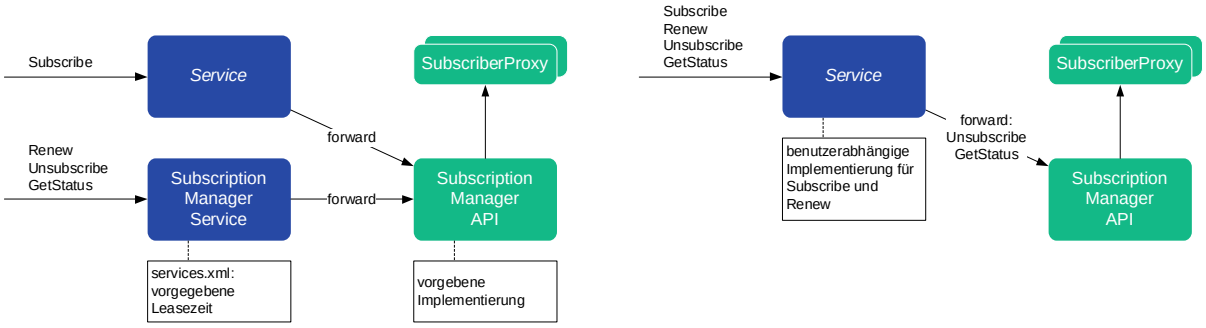

Abbildung 5.6: Nachrichtenfluss im Eventing-Modul für vorgegebene und benutzerspezifische Implementierung

enden. Um Push-basierte Ereignisse zu empfangen, muss der Dienstnutzer den Endpunkt einer *Event Sink* angeben.

5.1.5 Die DPWS-Erweiterung

DPWS ist ein Profil, welches vorhandene Web-Services-Spezifikationen benutzt. Dieses Prinzip wurde ebenfalls in der Softwareimplementierung umgesetzt. Zum einen durch Axis2, welches einen Mechanismus zur Verfügung stellt, mit dem zusätzliche Prokotolle in Form von Modulen installiert werden können, zum anderen durch das DPWS-Modul, welches die anderen Module nutzt und nur die Erweiterungen und Einschränkungen realisiert, die das Profil definiert.

Der größte Anteil der Erweiterungen durch das DPWS-Profil entfällt auf Discovery: Nicht alle Dienste, sondern nur Geräte sind *Target Services*, die Konstanten des Retransmission-Algorithmus der SOAP-over-UDP-Spezifikation wurden für DPWS-Netzwerke verändert, DPWS definiert den Aufbau der Geräte- und Dienstrepräsentationen und außerdem wird mit WS-Transfer ein weiteres Protokoll für Discovery benötigt. Diese Erweiterungen wurden mit dem DPWS-Modul für Axis2 umgesetzt. Weiterhin enthält die Bibliothek eine prototypische Anpassung des Axis2-Codegenerators, der optional verwendet werden kann. Der Codegenerator basiert auf XML-Templates und kann daher relativ leicht erweitert werden.

Listing 5.3: Konfiguration eines Gerätes in der services.xml

```
1 <serviceGroup>
2     <service name="AirConditionerDeviceService">
3         ...
4         <parameter name="Discovery" auto="false">
5             ...
6         </parameter>
```

```
7         ...
8         <parameter name="DPWSDevice">
9             <dpws:ThisDevice
10                xmlns:dpws="http://schemas.xmlsoap.org/ws/2006/02/devprof">
11                <dpws:FriendlyName xml:lang="EN">
12                    Air Conditioner device
13                </dpws:FriendlyName>
14                <dpws:FirmwareVersion>alpha</dpws:FirmwareVersion>
15                <dpws:SerialNumber>1</dpws:SerialNumber>
16            </dpws:ThisDevice>
17            <dpws:ThisModel
18                xmlns:dpws="http://schemas.xmlsoap.org/ws/2006/02/devprof">
19                <dpws:Manufacturer>WS4D</dpws:Manufacturer>
20                <dpws:ManufacturerUrl>
21                    http://www.ws4d.org/
22                </dpws:ManufacturerUrl>
23                <dpws:ModelName>Air Conditioner device</dpws:ModelName>
24                <dpws:ModelNumber>1.0</dpws:ModelNumber>
25                <dpws:ModelUrl>http://www.ws4d.org/</dpws:ModelUrl>
26                <dpws:PresentationUrl>
27                    http://www.ws4d.org/
28                </dpws:PresentationUrl>
29            </dpws:ThisModel>
30            <dpws:Host
31                xmlns:dpws="http://schemas.xmlsoap.org/ws/2006/02/devprof">
32                <dpws:ServiceID>acdevice-service-id:1</dpws:ServiceID>
33            </dpws:Host>
34            <dpws:Hosted name="ACService"
35                xmlns:dpws="http://schemas.xmlsoap.org/ws/2006/02/devprof">
36                <dpws:Types
37                xmlns:ac="http://www.ws4d.org/axis2/tutorial/AirConditioner">
38                    ac:ACService
39                </dpws:Types>
40                <dpws:ServiceId>
41                    http://ws4d.tutorialdevice.org/ACService1
42                </dpws:ServiceId>
43            </dpws:Hosted>
44        </parameter>
45    </service>
46</serviceGroup>
```

Auf der Dienstseite werden die Konfigurationen ausschließlich in der *services.xml* des Gerätedienstes vorgenommen. Dienste, die vom Gerät verwaltet werden, brauchen nicht zusätzlich konfiguriert zu werden. Listing 5.3 zeigt den Aufbau einer Gerätekonfiguration. Der Discovery-Parameter enthält die gleichen Einstellungen wie im Abschnitt 5.1.3 beschrieben, mit dem einzigen Unterschied, dass hier das automatische Discovery durch den *DiscoveryObserver* deaktiviert wurde und stattdessen durch das DPWS-Modul selbst durchgeführt wird. Dazu installiert das

DPWS-Modul einen *DpwsObserver*, der analog zum *DiscoveryObserver* arbeitet. Der *DpwsObserver* liest bei einer Geräteinstallation den *DPWSDevice*-Parameter aus der Konfigurationsdatei. Die dem Gerät zugehörigen Dienste werden ebenfalls zur Laufzeit über das *name*-Attribut der *dpws:Hosted*-Elemente ermittelt. Sämtliche Discovery-Logik wird vom DPWS-Modul, die für die Dienstrepräsentation durch den generierten Code übernommen.

Das DPWS-Modul besitzt zusammen mit dem generierten Code ein voreingestelltes Verhalten für *Subscription*-Nachrichten, die einen *Action Filter* (Abs. 2.8) spezifizieren. Die referenzierten Operationen werden zunächst auf Richtigkeit überprüft und dann im *SubscriberProxy* abgelegt. Tritt in einem Gerät ein Ereignis auf, werden alle Abonnenten, die die entsprechende Ereignisnachricht-Operation im Filter angegeben haben, benachrichtigt.

Dienstnutzer verwenden die *DiscoveryClientAPI* des Discovery-Moduls, um nach Geräten zu suchen und die *DPWSClientAPI*, um die Geräte- und Dienstrepräsentationen zu erhalten sowie Ereignisse zu abonnieren. Dazu stehen entsprechende statische Methoden zur Verfügung. Auf eine tiefergehende Erläuterung wird hier verzichtet, da sie bis auf wenige Änderungen weitestgehend den Funktionsweisen der zuvor vorgestellten Bibliotheken entsprechen.

5.2 Geräte- und Dienstvorlagen für DPWS

5.2.1 Motivation und Anforderungen

Um Geräte bzw. Dienste zu finden, werden sowohl in UPnP als auch in DPWS *Typen* verwendet. Ein solcher Typ substituiert in der Regel ein Anwendungsprotokoll. Kennt sowohl der Dienstnutzer als auch der Dienst den gleichen Typ (und damit das Anwendungsprotokoll), so können sie miteinander kommunizieren bzw. sich *verstehen*.

Es hat sich als vorteilhaft erwiesen, Typen formal in Dokumenten zu spezifizieren, da anschließend aus den Dokumenten mit Codegeneratoren Code erzeugt werden kann. Für UPnP werden beispielsweise Geräte- und Diensttypen innerhalb des UPnP Forums in einem festgelegten Prozess (DCP Specification Process) definiert und veröffentlicht. Neben einem beschreibendem Dokument gibt es jeweils sogenannte *Device Templates* bzw. *Service Templates*, die die Typen formal spezifizieren. Es existieren verschiedene UPnP-Stacks, die aus diesen formalen Dokumenten Code erzeugen können (z. B. Intel UPnP SDK[3]).

DPWS fehlt ein entsprechender Mechanismus. Es gibt weder ein Format (Vorlagensystem), um Typen zu spezifizieren, noch ein Gremium oder einen standardisierten Prozess, welche diese Aufgabe umsetzen. Daher soll im Folgenden eine Lösung für dieses Problem für DPWS vorgestellt werden. Diese soll folgende Anforderungen erfüllen:

[3] `http://www.intel.com/cd/ids/developer/asmo-na/eng/downloads/upnp/tools/index.htm`

- **Einfachheit** Mit dem Vorlagensystem soll die Erstellung von Geräte- und Diensttypen möglichst einfach zu bewerkstelligen sein. Beispielsweise sollen keine speziellen Werkzeuge vorausgesetzt werden müssen. Die Vorlagen sollen in einem Textformat verfasst werden und intuitiv verständlich (lesbar) sein.
- **Codegenerierung** Ein großer Vorteil von Web Services besteht darin, direkt aus der WSDL Code erzeugen zu können, der die zugrundeliegenden Protokolle vor dem Entwickler verbirgt (Zeitersparnis, geringere Fehleranfälligkeit usw.). Dieses Prinzip soll bei den Typdokumenten fortgesetzt werden, indem es möglich ist, Anwendungs- und Implementierungscode für Geräte und Dienste zu generieren.
- **Versionierung** Die Vergangenheit zeigt, dass Schnittstellen und Datenformate im Laufe der Zeit weiterentwickelt werden und sich demzufolge ändern. Um ähnliche Dienste möglichst lange verwenden zu können, müssen sie abwärtskompatibel zu älteren Versionen sein. Das Typisierungssystem soll daher einen Versionierungsmechanismus mitbringen, der auch beim Discovery berücksichtigt werden kann.
- **Formalismus** Die Spezifizierung muss formal ausgedrückt werden können, damit sie maschinenlesbar und verarbeitbar ist. Nur so kann die Anforderung an die Codegenerierung erfüllt werden.
- **Erreichbarkeit** Da Geräte- und Diensttypen einen globalen Gültigkeitsbereich besitzen können und außerdem verteilt definiert und genutzt werden sollen, sollten sie über das Internet erreichbar sein.
- **Dienstinstanzen** Geräte benötigen u. U. mehrere typgleiche Dienste. Das Vorlagensystem muss daher zwischen unterschiedlichen Dienstinstanzen unterscheiden können.

5.2.2 Grundidee

Die Dienste eines DPWS-Gerätes sind herkömmliche Web Services, deren Schnittstellen mittels WSDL beschrieben werden. WSDL verwendet das Konzept der *qualifizierten Namen* (*qualified names*, QName) von XML, um zusammengehörige Operationen als Port-Type auf einen eindeutigen Bezeichner abzubilden und an anderen Stellen referenzieren zu können. QNames besitzen einen Namensraum (URI) und einen lokalen Namen.

Bei der mittels WS-Discovery durchgeführten Geräte- und Dienstsuche werden in DPWS ebenfalls QNames für Geräte- und Diensttypen verwendet. In der Spezifikation ist die Bedeutung dieser jedoch völlig offengelassen. Die Grundidee der vorgestellten Lösung ist daher, diese Typen (QNames) auf die QNames der WSDL-Port-Types abzubilden.

Typvorlagen werden in XML notiert, da dieses die vorherrschende Auszeichnungssprache im Web ist und DPWS sie bereits durchgehend verwendet. Im Anhang A ist ein entsprechendes

XML-Schema zu finden, welches für die Validierung von Typvorlagen verwendet werden kann. Um die Vorlagen verständlicher zu gestalten, sind die lokalen Namen einiger XML-Elemente an diejenigen in WS-Discovery angelehnt. Diese Wiederverwendung bezieht sich lediglich auf die Bedeutung der Elemente im weitesten Sinne. Der Namensraum ist jedoch ein anderer, weshalb es sich trotz gleicher Namen um unterschiedliche Elemente handelt. Weiterhin wird analog zu UPnP zwischen Typvorlagen für Geräte und für Dienste unterschieden. Dadurch können Diensttypen unabhängig von Geräten standardisiert und für verschiedene Geräte wiederverwendet werden. Beispielsweise kann damit ein *PowerService*, welcher Geräte in verschiedene Modi setzen kann (an, aus, schlafend usw.), in jeder Art Gerät verwendet werden. Andere Beispiele für geräteunabhängige Dienste sind Verwaltungs- und Berechtigungsdienste.

5.2.3 Typisierung von Diensten

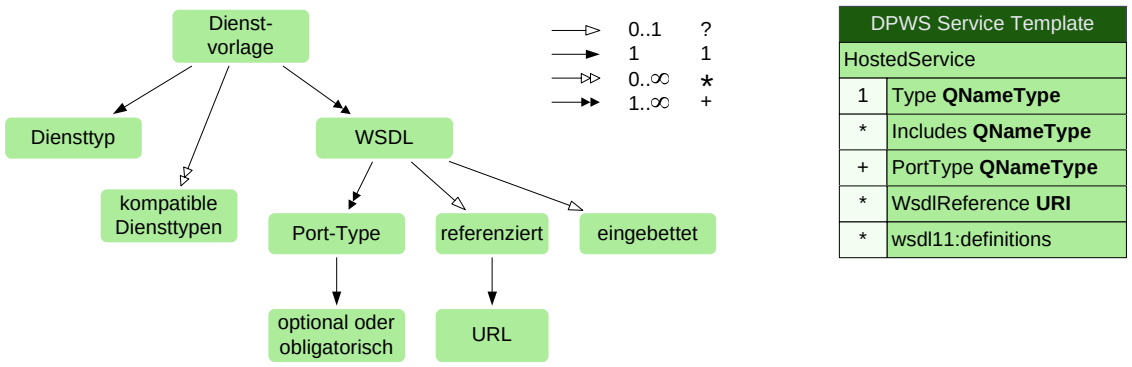

Abbildung 5.7: Struktur der Dienstvorlagen

Abbildung 5.7 zeigt die Struktur einer Dienstvorlage. Sie definiert genau einen Diensttyp als QName und kann beliebig viele Diensttypen aufzählen, zu welchem der Diensttyp kompatibel sein muss. Über diesen Mechanismus kann die Versionierung realisiert werden. Für die Abbildung auf QNames der Port-Types, die der Diensttyp implementieren muss, können ein oder mehrere WSDL-Dokumente mit einem Link referenziert oder direkt eingebettet werden. Das XML-Schema für die Dienstvorlagen setzt die dargestellte Struktur um.

Listing 5.4 enthält ein konkretes Beispiel, welches einen *PowerService* definiert und ihn an den Typ *ws4dp:Power2.0*[4] bindet. Die Vorlagen verwenden eine eigene Notation für QNames, die sich von der häufig verwendeten (u. a. in WS-Discovery) vereinfachten Notation dahingehend unterscheidet, dass sie nicht als Textknoten notiert ist, sondern eigene Elemente für URI und

[4]Das Präfix *ws4dp* steht hier für die Namespace-URI *http://www.ws4d.org/templates/power*

lokalen Namen verwendet. Derartige Dokumente lassen sich einfacher parsen und sind vor allem für XSL-Transformer besser verarbeitbar, da keine proprietären Algorithmen zum Extrahieren der Namespaces implementiert werden müssen.

Listing 5.4: Beispiel für eine Dienstvorlage

```
<?xml version="1.0" encoding="UTF-8"?>
<t:HostedService xmlns:t="http://www.ws4d.org/templates/">
    <t:Type>
        <t:URI>http://www.ws4d.org/templates/power</t:URI>
        <t:localName>Power2.0</t:localName>
    </t:Type>
    <t:Includes>
        <t:URI>http://www.ws4d.org/templates/power</t:URI>
        <t:localName>Power1.0</t:localName>
    </t:Includes>
    <t:PortType>
        <t:URI>http://www.ws4d.org/templates/power</t:URI>
        <t:localName>PowerPortType</t:localName>
    </t:PortType>
    <t:WsdlReference>
        http://www.ws4d.org/templates/power/Power2.0.wsdl
    </t:WsdlReference>
</t:HostedService>
```

Mit Dienstvorlagen können Diensttypen (QNames) auf bestimmte WSDL-Port-Types abgebildet werden. Ein Typ kann dabei beliebig viele Port-Types verwenden. Da Port-Types in unterschiedlichen WSDL-Dokumenten und damit an verschiedenen Orten definiert sein können, ist in Dienstvorlagen beides möglich: Referenzieren einer WSDL mittels einer URI innerhalb eines *WsdlReference*-Elementes sowie die direkte Einbettung einer WSDL in Form eines herkömmlichen *wsdl11:definitions*-Elementes.

Das *PortType*-Element listet die obligatorischen Port-Types als QNames auf, die in einer Implementierung vorhanden sein müssen. Dienstnutzer, die diesen Diensttyp verwenden wollen, können in der Folge davon ausgehen, dass die entsprechenden Operationen tatsächlich implementiert sind. Nicht aufgeführte Port-Types werden als optional betrachtet. Dadurch lassen sich herstellerseitig zusätzliche Funktionalitäten integrieren, ohne die eigentliche Typdefinition zu verletzen. Außerdem ermöglicht diese Herangehensweise eine bessere Verständlichkeit, da die erforderlichen Interfaces (Port-Types) explizit aufgelistet werden.

Um die Weiterentwicklung von Diensten zu ermöglichen, die Abwärtskompatibilität jedoch zu bewahren, kann mit dem *Includes*-Element ein Diensttyp referenziert werden, der im neu definierten Diensttyp vollständig enthalten ist. Es können beliebig viele solcher Elemente verwendet werden.

Die bis hierher vorgestellten Konzepte ermöglichen eine stufenweise aufeinander aufbauende Typdefinition: Beispielsweise enthält ein WSDL-Dokument die Port-Types *P1*, *P2* und *P3*. Dann kann ein allgemeiner Dienst *S1* lediglich den Port-Type *P1* erfordern, während ein zweiter, spezifischerer Dienst *S2* den Dienst *S1* inkludieren und zusätzlich die Port-Types *P2* und *P3* als erforderlich kennzeichnen kann.

5.2.4 Typisierung von Geräten

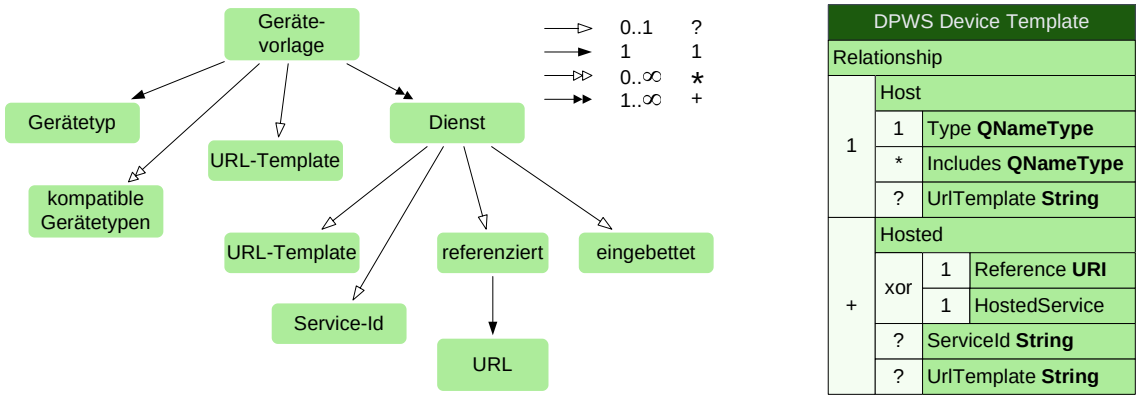

Abbildung 5.8: Struktur der Gerätevorlagen

Abbildung 5.8 zeigt die Struktur einer Gerätevorlage. Sie definiert analog zur Dienstvorlage genau einen Gerätetyp als QName und kann beliebig viele Gerätetypen enthalten, zu welchem der Gerätetyp kompatibel sein muss. Weiterhin können ein oder mehrere Diensttypen referenziert oder eingebettet werden, die der Gerätetyp implementieren muss. Listing 5.5 enthält ein konkretes Beispiel, welches einen Typ für eine Webcam definiert.

DPWS verwendet das Konzept der *Relationships* für Dienste, die logisch einem Gerät zugeordnet sind. Ein Relationship enthält stets ein *Host*-Element, welches das Gerät referenziert, für welches das Relationship gilt, und für jeden in logischer Beziehung stehenden Dienst ein *Hosted*-Element. Gerätevorlagen verwenden die Begriffe analog. Innerhalb eines *Host*-Elements wird ein Diensttyp deklariert. Hier können wiederum *Include*-Elemente verwendet werden, um abwärtskompatible Gerätetypen zu kennzeichnen.

Für jeden zu implementierenden Dienst (*Dienstinstanz*) wird ein *Hosted*-Element notiert. Dieses enthält entweder eine Referenz zu einer existierenden Diensttypdefinition in Form einer URI, oder aber die Deklaration wird direkt mit einem *HostedService*-Element eingebettet. Dieses entspricht dem Aufbau einer Dienstvorlage, wie oben beschrieben. Jeder Dienst kann außerdem

Listing 5.5: Beispiel für eine Gerätevorlage

```
<?xml version="1.0" encoding="UTF-8"?>
<t:Relationship xmlns:t="http://www.ws4d.org/templates/">
    <t:Host>
        <t:Type>
            <t:URI>http://www.ws4d.org/templates/webcam</t:URI>
            <t:localName>Webcam</t:localName>
        </t:Type>
        <t:UrlTemplate>http://{ip}:4672/webcam</t:UrlTemplate>
    </t:Host>
    <t:Hosted>
        <t:Reference>
            http://www.ws4d.org/templates/power/PowerService2.0.xml
        </t:Reference>
        <t:ServiceId>
            http://www.ws4d.org/power/powerservice2
        </t:ServiceId>
    </t:Hosted>
    <t:Hosted>
        <t:Reference>
            http://www.ws4d.org/templates/config/BasicDeviceConfig2.1.xml
        </t:Reference>
    </t:Hosted>
    <t:Hosted>
        <t:Reference>
            http://www.ws4d.org/templates/webcam/Webcam1.0.xml
        </t:Reference>
        <t:UrlTemplate>http://{ip}:4672/webcam/service</t:UrlTemplate>
    </t:Hosted>
</t:Relationship>
```

mit einer Service-ID markiert werden. Dadurch lassen sich Dienste bei der Verwendung mehrerer typgleicher Dienste voneinander unterscheiden, außerdem benötigt DPWS diesen Parameter für die Beschreibung.

Für Gerätetypen und für Dienstinstanzen, die bei der Deklaration eines Gerätetyps verwendet werden, können URL-Templates angegeben werden. Diese werden verwendet, um statische bzw. dynamische URL-Segmente für Gerätetypen bzw. Dienstinstanzen festzulegen. Im Beispiel ist der Port des Webcam-Dienstes auf den Port *4672* festgelegt und auch der URL-Pfad ist statisch vorgegeben. In diesem Fall reicht die Kenntnis der IP-Adresse aus, um den Dienst ansprechen zu können. Dadurch kann die Discovery-Phase wesentlich verkürzt werden, da keine Geräte- und Dienstbeschreibungen transferiert werden müssen. Außerdem wird dadurch die durch Discovery verursachte Netzwerklast verringert. Dynamische URL-Teile werden in Anlehnung an [136] in geschweiften Klammern notiert.

5.3 Klassifizierung von SOA-Teilnehmern

Bei der Umsetzung einer SOA kommt dem Dienstentwurf eine wichtige Bedeutung zu. Ähnlich wie beim objektorientierten oder komponentenbasierten Entwurf entscheidet der *Serviceschnitt* über die langfristige Wiederverwendbarkeit der Dienste und der Flexibilität der gesamten SOA. Während sich bei der Objektorientierung längst Begriffe bzw. Pattern durchgesetzt haben, nach denen sich Klassen in Kategorien hinsichtlich Verhalten oder Struktur einteilen lassen [137], fehlt eine entsprechende Klassifizierung für Dienste bzw. für SOA-Teilnehmer.

SOA-Teilnehmer	Icon	SOA-Teilnehmer	Icon
Fassade		Öffentlich	
Adapter	X Y	Gerät	
Gateway		Anwendung	
Zusatz		Fremd	
Basis		SOA Basis	S A
Prozess			

Abbildung 5.9: Klassifizierung von SOA-Teilnehmern

Abbildung 5.9 zeigt verschiedene Arten von SOA-Teilnehmern[5]. Sie sind aus [27] übernommen worden und wurden hier um die Teilnehmer *Gerät*, *Anwendung* und *SOA Basis* ergänzt. *Basisdienste* sind grundlegende Dienste und immer unabhängig von anderen Diensten. Sie kapseln in der Regel Daten und deren Operationen. Für *SOA-Basisdienste* gilt das gleiche, jedoch sind sie eng an die verwendete SOA-Technologie gebunden und unterstützen bzw. ermöglichen erst die SOA. *Geräte* sind prinzipiell auch Basisdienste, jedoch beziehen sie sich explizit auf real existierende (eingebettete) Geräte. Normalerweise geht aus der Struktur einer SOA bzw. aus der Anordnung ihrer Dienste nicht hervor, wie die darunterliegenden Softwarekomponenten organisiert sind. *Geräte* bilden hier als einzige Dienstklasse eine Ausnahme. Bei der Erstellung eines SOA-Diagramms muss also für jedes eingebettete Gerät jeweils ein *Gerät* als Dienst abgebildet werden.

Ein *Adapter* ist ein Dienst, der eine an einen bestimmten Dienstnutzer angepasste Schnittstelle anbietet. Zeitlich betrachtet existiert der Dienstnutzer also, bevor der Adapter eingesetzt wird. Der Adapter übersetzt dazu die vom Dienstnutzer erwarteten Datenstrukturen und Signaturen

[5] Bezeichnungen und Icons sind nicht standardisiert.

in die des gekapselten Dienstes und umgekehrt. *Fassaden* kapseln mehrere Dienste und bieten eine völlig neue Sicht auf diese an. Fassaden können eingeschoben werden, um Dienstnutzern die Komplexität der gekapselten Dienste zu entziehen. Schließlich erweitert ein *Zusatz*-Dienst einen anderen Dienst um neue Operationen.

Gateways kapseln Dienste, die technologisch inkompatibel zu der verwendeten SOA-Technologie sind. Oftmals fallen darunter Altsysteme, die auf diesem Weg in die SOA integriert werden, oder aber Teilnehmer, die zu einer technologisch anders realisierten SOA gehören. In beiden Fällen werden diese Randkomponenten als *Fremd*-Dienste bezeichnet.

Prozesse bilden Geschäftsprozesse als Dienst ab und verwenden für ihre Ausführung andere Dienste. Als *öffentlich* wird ein Dienst bezeichnet, wenn er für die Verwendung über die eigene SOA hinaus entworfen wurde. *Anwendungen* sind reine Dienstnutzer, bieten selbst also keine Dienste an.

Oftmals sind Dienste Mischformen der genannten Klassifizierungen. Beispielsweise könnte ein öffentlicher Dienst mehrere interne Dienste verwenden und gleichzeitig neue Funktionalitäten anbieten, die für fremde Dienstnutzer adressiert sind. In diesem Fall ist der Dienst gleichzeitig Fassade und ein Zusatzdienst.

Eine Klassifizierung von SOA-Teilnehmern hat viele Vorteile, da aus ihr beispielsweise die Komplexität, die Wiederverwendbarkeit oder der Implementierungsaufwand von Diensten hervorgehen kann. Die hier vorgestellte Klassifizierung wird im nachfolgenden Kapitel verwendet, um die vertikalen Abhängigkeiten (Dienststruktur) zwischen Diensten und dem Dienstzweck visuell darzustellen. Die Darstellung wird hier *SOA-Diagramm* genannt.

5.4 Serviceorientierter Aufbau einer Lokalisierungsplattform mit DPWS

Im Folgenden geht es um den serviceorientierten Aufbau einer Lokalisierungsplattform mit dem Devices Profile for Web Services. Das Kapitel verfolgt zwei Ziele: Zum einen soll eine Lokalisierungsplattform beschrieben werden, die heterogene Lokalisierungssysteme in eine SOA integriert und als Dienste abbildet. Zum anderen sollen anhand dieser Anwendungsdomäne die Ideen, die hinter SOA, DPWS und dem im Abschnitt 5.2 vorgestellten Typisierungsmechanismus stehen, verdeutlicht werden. Zu diesem Zweck wird ein Szenario beschrieben, welches sich fiktiv über einen sehr langen Zeitraum (mehrere Jahre) erstreckt. Es ist in mehrere Phasen unterteilt. In jeder Phase werden die neuen Anforderungen an das System formuliert und die Umsetzung für die SOA besprochen. Dabei wird die im Abschnitt 5.3 eingeführte Klassifizierung für SOA-Diagramme verwendet. Außerdem werden Auszüge aus den entsprechenden WSDL- und XML-Schema-Dokumenten sowie Algorithmen gezeigt, so wie sie auch für die praktische Evaluierung

Tabelle 5.1: WSDL-Operationen des BlueScan-Gerätes

Operation	MEP	Parameter
SetConfig	Request-Response	In: Die Zeitdauer zwischen zwei Inquiries (*t.interval*) und Zeitdauer zwischen Inquiry und ScanResult (*t.block*), Out: Die akzeptierten (gleichen) Parameter
ScanResult	Notification	Zeitpunkt des Inquiries und Liste der Bluetooth-Geräteadressen

angewandt wurden. Die vollständigen Dokumente befinden sich im Anhang B. Das Szenario spielt in einer Firma namens *Socom* (*Service-oriented company*). Ein Großteil der vorgestellten Dienste wurde praktisch implementiert. Darauf wird im anschließenden Kapitel eingegangen.

5.4.1 Phase I: BlueScan

Socom möchte ein System installieren, mit welchem die aktuellen Positionen ihrer Mitarbeiter bestimmt werden können. Die Lösung soll kostengünstig sein. Außerdem ist eine grobe Genauigkeit ausreichend. Socom entschließt sich von Anfang an, ihr System serviceorientiert aufzubauen, um die Investitionen langfristig zu sichern. Die Wahl fällt auf das *BlueScan-System*.

BlueScan besteht aus einem *BlueScan-Server* und beliebig vielen *BlueScan-Geräten* (*BlueScan-Devices*). Die BlueScan-Geräte werden an festen Orten montiert und suchen in regelmäßigen Abständen nach Bluetooth-Geräten, die sich in der Nähe befinden. Dazu senden sie periodisch eine Bluetooth-Inquiry-Nachricht aus. Geräte, die sich in Reichweite befinden, antworten auf diese Anfrage. Die Antwort enthält u. a. die jeweilige 6 Byte lange Bluetooth-Geräteadresse, die das antwortende Gerät (und damit den Mitarbeiter) eindeutig identifiziert (z. B. *00:60:57:39:E7:AE*).

Listing 5.6: Algorithmus auf dem BlueScan-Gerät (informal)

```
1 while(true){
2     if(t_interval > 0){
3         start_time = current_time
4         addresses = send_inquiry(t_block)
5         dpws_notify("ScanResult", addresses, start_time)
6         sleep(t_interval - t_block)
7     }
8 }
```

Das BlueScan-Gerät ist ein reguläres DPWS-Device. Es enthält als einzigen Dienst den *BlueScan-Service*, welcher die beiden Operationen *SetConfig* und *ScanResult* anbietet (Tabelle 5.1).

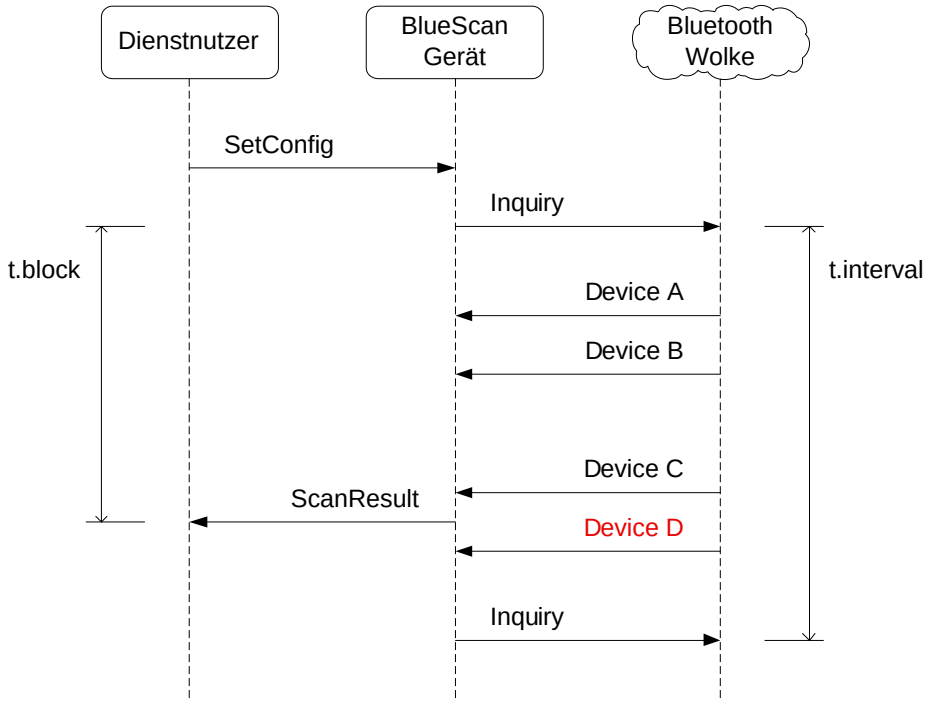

Abbildung 5.10: Sequenzdiagramm für die Konfiguration, Inquiry und Ergebnisübertragung auf einem BlueScan-Gerät

Die Funktionsweise ist im Sequenzdiagramm (Abbildung 5.10) dargestellt. Die *SetConfig*-Operation wird verwendet, um das Gerät zu konfigurieren. Hier kann die Länge zwischen zwei Inquiry-Nachrichten (*t.interval*) sowie die Verzögerungszeit (*t.block*) vom Inquiry-Nachricht bis zum Versenden der Ergebnisse angegeben werden. Ein Intervall von *0* verhindert das Aussenden von Inquiry-Nachrichten, ansonsten arbeitet der Scanner kontinuierlich.

Die *ScanResult*-Operation ist als Notification-MEP modelliert. Der Scanner wartet nach einem Inquiry die in *t.block* angegebene Zeit und speichert währenddessen die eintreffenden Adressen (Zeile 4, Listing 5.6). Nach Ablauf der Verzögerungszeit wird das Ergebnis an alle Abonnenten übermittelt (Zeile 5). *ScanResult* enthält den Zeitpunkt des Inquiries sowie eine Adressliste. Die Wahl einer geeigneten Verzögerungszeit stellt einen Trade-off zwischen zeitnahen Messergebnissen und der ermittelten Gerätemenge dar. Später eintreffende Antworten werden nicht registriert (Device D in Abbildung 5.10).

Damit der BlueScan-Server die Scanner später mittels WS-Discovery finden kann, wird hier ein spezieller Diensttyp *BlueScanScanService* deklariert, der die entsprechende WSDL referenziert (Listing 5.7), sowie ein Gerätetyp *BlueScanDevice1.0*, der als einzigen Dienst den Blue-

ScanScanService referenziert (Listing 5.8).

Listing 5.7: BlueScan-ScanService-Typ

```
<?xml version="1.0" encoding="UTF-8"?>
<t:HostedService xmlns:t="http://www.ws4d.org/templates/">
    <t:Type>
        <t:URI>http://www.ws4d.org/bluescan</t:URI>
        <t:localName>BlueScanScanService</t:localName>
    </t:Type>
    <t:PortType>
        <t:URI>http://www.ws4d.org/bluescan</t:URI>
        <t:localName>BlueScanScanPortType</t:localName>
    </t:PortType>
    <t:WsdlReference>
        http://www.ws4d.org/specs/bluescan/bluescan-scan-service.wsdl
    </t:WsdlReference>
</t:HostedService>
```

Für die Lokalisierung der Bluetooth-Geräte (bzw. der Mitarbeiter) müssen ständig alle vorhandenen BlueScan-Geräte beobachtet werden. Fallen Scanner aus oder werden neue montiert, müssen sie neu konfiguriert und ausgewertet werden. Da die Lokalisierung voraussichtlich ohnehin von mehreren Teilnehmern genutzt werden wird, wird für diese Aufgabe ein weiterer Dienst entworfen – der BlueScan-Server. Dieser ist gleichzeitig Dienstnutzer (für die Scanner) und Dienst. Für einen Nutzer erscheint der BlueScan-Server vollkommen transparent, da er die dahinterliegende Infrastruktur (Topologie und Anzahl von BlueScan-Geräten) verbirgt.

Listing 5.8: BlueScan-Device-Typ

```
<?xml version="1.0" encoding="UTF-8"?>
<t:Relationship xmlns:t="http://www.ws4d.org/templates/">
    <t:Host>
        <t:Type>
            <t:URI>http://www.ws4d.org/bluescan</t:URI>
            <t:localName>BlueScanDevice1.0</t:localName>
        </t:Type>
    </t:Host>
    <t:Hosted>
        <t:Reference>
            http://www.ws4d.org/specs/bluescan/bluescan-scan-service-template.xml
        </t:Reference>
        <t:ServiceId>
            http://www.ws4d.org/bluescan/scanservice
        </t:ServiceId>
    </t:Hosted>
</t:Relationship>
```

Bisher liefern die Scanner lediglich Adressen zurück. Um Aussagen über die Positionen zu

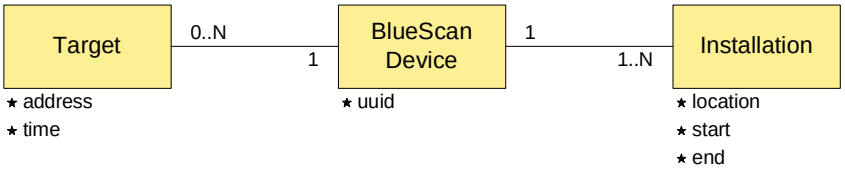

Abbildung 5.11: Entity-Relationship-Modell für den BlueScan-Server

Tabelle 5.2: WSDL-Operationen des BlueScan-Servers

Operation	MEP	Parameter
Scan	Request-Response	In: Den zu scannenden Ort (*location*) und die Zeitdifferenz, aus der sich ergibt, wie alt das Messergebnis maximal sein darf (*t.offset*), Out: Liste der Bluetooth-Geräteadressen und ihren Zeiten
Find	Request-Response	In: Die zu suchende Bluetooth-Geräteadresse (*address*) und maximale Zeitdifferenz (*t.offset*), Out: Die gefundenen Orte (*locations*)

erlangen, müssen die installierten Scanner mit Orten assoziiert werden. Der Einfachheit halber werden einfache aussagefähige Bezeichnungen, z. B. *Flur im Sektor 12*, verwendet. Die Zuweisung erfolgt auf dem BlueScan-Server und wird auch dort gespeichert. Abbildung 5.11 zeigt das zugehörige Entity-Relationship-Modell (ERM), welches für eine Datenbank verwendet werden kann. Darin entspricht eine *Installation* einer solchen Abbildung von einem BlueScan-Gerät auf einen Ort. Ihre Gültigkeit ist durch eine Anfangs- und Endzeit begrenzt. Dadurch können BlueScan-Geräte beispielsweise infolge einer Reorganisation an neue Orte verschoben werden, ohne dass die alten Daten ungültig werden. Die Schnittstelle zur Einstellung dieser Orte wird hier nicht weiter betrachtet, sie ist auch nicht Teil der WSDL.

Ein *BlueScan-Device* repräsentiert einen Scanner. Um diesen eindeutig zu identifizieren, wird seine UUID gespeichert. DPWS schreibt vor, dass die UUID eines Gerätes für die gesamte Lebenszeit konstant sein muss. Die UUID bietet einen weiteren Vorteil gegenüber der Alternative, bei welcher die Bluetooth-Geräteadresse des Scanners gespeichert werden würde: Muss das Bluetooth-Modul aufgrund eines Defektes einmal ausgetauscht werden, ändert sich auch die Bluetooth-Geräteadresse, während die UUID erhalten bleibt. Ein *Target* steht schließlich für ein Messergebnis, bei welchem die gescannte Bluetooth-Geräteadresse zusammen mit der Zeit gespeichert wird.

Listing 5.9: Algorithmus der Scan-Operation auf dem BlueScan-Server (informal)

```
1 scan(location, t_offset){
2     time = current_time
3     installation = find_current_installation(location, time)
4     bluescan_device = installation.getScanner
5     targets = bluescan_device.getTargets(time, t_offset)
6     return targets
7 }
```

Der BlueScan-Server bietet zwei Operationen an. Mit *Scan* lassen sich analog zum BlueScan-Gerät Orte scannen mit dem Unterschied, dass hier die Ortsrepräsentation und die Zeit, die die Messergebnisse maximal alt sein dürfen, als Eingangsparameter fungieren. Der Algorithmus (Listing 5.9) sucht zunächst nach dem BlueScan-Gerät, welches aktuell dem angegebenen Ort zugewiesen ist (Zeilen 3–4). Danach werden alle gescannten Geräte ermittelt, deren Messzeitpunkte innerhalb der Offsetgrenze $t.offset$ liegen (Zeile 5). Der Aufruf kann sofort zurückkehren, da nur die Datenbank durchsucht werden muss.

Listing 5.10: Algorithmus der Find-Operation auf dem BlueScan-Server (informal)

```
1  find(address, t_offset){
2      time = current_time
3      targets = find_targets(address, time, t_offset)
4      locations = new array
5      for each target in targets{
6          target_time = target.getTime
7          locations.add(
8              target.getScanner.getInstallationForTime(target_time).getLocation,
9              target_time)
10     }
11     return locations
12 }
```

Mit *Find* lässt sich nach einer Bluetooth-Geräteadresse suchen. Das Ergebnis gibt eine Liste der gefundenen Orte zurück. Der Algorithmus (Listing 5.10) filtert zunächst aus allen Messergebnissen diejenigen heraus, die der angegebenen Bluetooth-Geräteadresse entsprechen und innerhalb der Zeitgrenze liegen (Zeile 3). Über diese Liste wird iteriert (Zeilen 5–10) und für jeden Eintrag der Ort und die Zeit ermittelt (Zeilen 8–9).

Drei weitere Algorithmen müssen betrachtet werden. Der BlueScan-Server arbeitet, wie bereits erwähnt, sowohl als Dienst als auch als Dienstnutzer. Um die Ergebnisse von den BlueScan-Geräten zu erhalten, muss er diese finden, sie konfigurieren und die Ergebnisse abonnieren. Das Listing 5.11 zeigt den Initialisierungscode, der aufgerufen wird, wenn der Server gestartet wird, und den Registrierungsalgorithmus. Bei der Initialisierung wird ein XML-Qualified-Name aus

Listing 5.11: Algorithmus der Registrierung von BlueScan-Geräten (informal)

```
1  namespace = "http://www.ws4d.org/bluescan"
2  local_name = "BlueScanDevice1.0"
3  device_type = new qname(namespace, local_name)
4  callback = register
5  dpws_middleware.listen(device_type, callback)
6
7  register(dpws_device){
8      if(!bluescan_devices.contain(dpws_device.getUUID)){
9          start_time = current_time
10         db_insert(new bluescan_device(dpws_device))
11         db_insert(new installation("unknown", start_time))
12         scan_service = dpws_device.getScanService
13         scan_service.subscribe("ScanResult")
14         scan_service.setConfig(30, 10)
15     }
16 }
```

der URI und dem lokalen Namen zusammengesetzt, der in Listing 5.8 als Gerätetyp deklariert wurde (Zeilen 1–3). Danach wird die DPWS-Middleware angewiesen, nach Geräten des entsprechenden Typs zu suchen und auch weiterhin abzuhören (Zeile 5), wobei der Registrierungsalgorithmus als Callback[6] gesetzt wird (Zeile 4).

Jede *ProbeMatch*-Antwort bzw. jede *Hello*-Nachricht führt zu einem Aufruf der *register*-Funktion durch die DPWS-Middleware. Es wird überprüft, ob bereits ein BlueScan-Gerät mit der entsprechenden UUID registriert ist (Zeile 8). Falls das nicht der Fall ist, wird ein neuer Datensatz für das Gerät und eine voreingestellte Ortsbezeichnung *unknown* angelegt (Zeilen 10–11). Anschließend wird die *ScanResult*-Operation abonniert (Zeile 13) und das BlueScan-Gerät mit einem Intervall von 30 s und einer Verzögerungszeit von 10 s konfiguriert (Zeile 14).

Listing 5.12: Algorithmus zum Empfangen von ScanResult-Ereignissen (informal)

```
1  receive_scan_result(dpws_device, addresses, inquiry_start_time){
2      bluescan_device = find_bluescan_device(dpws_device.getUUID)
3      for each address in addresses{
4          db_insert(new target(bluescan_device, address, inquiry_start_time))
5      }
6  }
```

Alle abonnierten BlueScan-Geräte schicken im eingestellten Intervall *ScanResult*-Ereignisse an den Server (Listing 5.12). Der Algorithmus ermittelt das BlueScan-Gerät, von welchem das Ereignis ausging (Zeile 2). Danach iteriert er über alle gescannten Geräteadressen (Zeilen 3–5) und legt für jede Adresse ein neues Messergebnis an (Zeile 4).

Die Abbildung 5.12 zeigt das SOA-Diagramm, welches in Phase I erarbeitet wurde. Die Blue-

[6] Referenz auf eine Funktion, die zu einem späteren Zeitpunkt von der Middleware aufgerufen wird

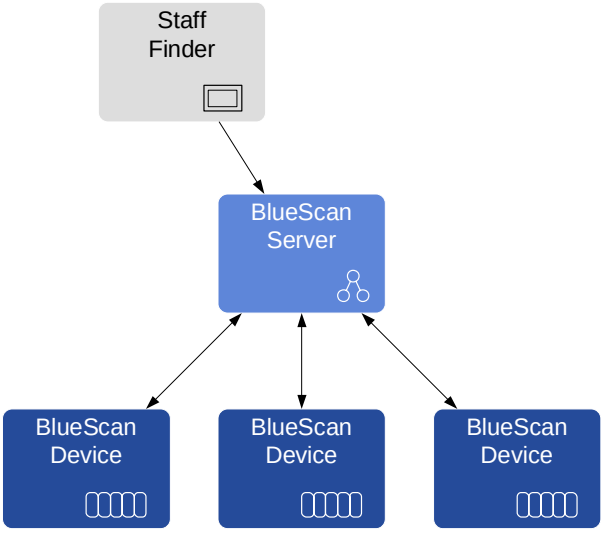

Abbildung 5.12: SOA-Diagramm für BlueScan-Geräte und BlueScan-Server

Scan-Devices entsprechen realen eingebetteten Geräten. Der BlueScan-Server fungiert dagegen als Service-Fassade. Er reduziert die Komplexität, indem er die (sich dynamisch ändernde) Anzahl von BlueScan-Geräten sowie deren Topologie vor anderen Dienstnutzern verbirgt. Weiterhin versteckt er den auf Ereignissen basierenden Mechanismus der Übertragung der Messergebnisse. *Staff Finder* stellt exemplarisch eine Anwendung dar, die den BlueScan-Server-Dienst nutzt (*Find*), um Mitarbeiter zu suchen.

5.4.2 Phase II: BlueScan2 – Eine neue BlueScan-Generation

Das BlueScan-Lokalisierungssystem wird nun seit einiger Zeit erfolgreich betrieben. Doch schon bald identifiziert Socom einige Schwächen, die das System mit sich bringt:

- **Trade-off der Verzögerungszeit** Die Verzögerungszeit (*t.block*) muss niedrig gewählt werden, damit die Ergebnisse schnell versendet werden können und somit die Lokalisierung von hoher Qualität (Aktualität) sein kann. Auf der anderen Seite muss die Verzögerungszeit hoch gewählt werden, um vollständige Ergebnisse zu erhalten.
- **Ungenaue Zeit** Der Messzeitpunkt wird auf den Zeitpunkt der Inquiry-Nachricht für alle Messergebnisse gesetzt. Der tatsächliche Messzeitpunkt kann jedoch später (bei über 10 Sekunden) liegen, wodurch die Genauigkeit stark beeinträchtigt wird.

Socom entschließt sich daher, eine neue Generation von BlueScan-Geräten als Version 2 zu entwickeln. Diese sollen folgende Anforderungen erfüllen:

- **Schnellere Antwortzeiten** Schnell antwortende Geräte sollen auch schnell an den Server übermittelt werden können.
- **Zeitgenauigkeit** Es sollen die tatsächlich gemessenen Zeiten erfasst werden.
- **Abwärtskompatibilität** Die BlueScan2-Geräte sollen entwickelt und nacheinander in Betrieb genommen werden, ohne das vorhandene System zu stören. Vor allem muss der BlueScan-Server mit den neuen Geräten weiterhin funktionieren. BlueScan2 soll also abwärtskompatibel zu BlueScan1 sein.

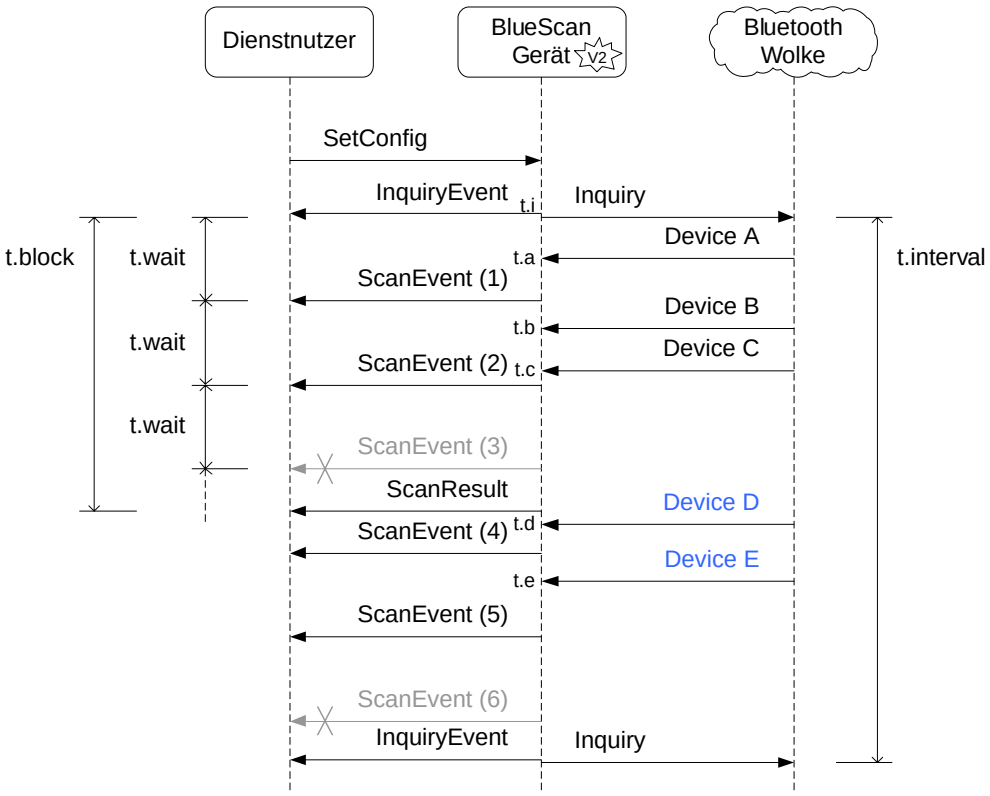

Abbildung 5.13: Sequenzdiagramm für die Konfiguration, Inquiry und Ereignisse eines BlueScan2-Gerätes

Abbildung 5.13 zeigt sequenziell die Arbeitsweise der BlueScan2-Geräte. Wichtigster Unterschied zur älteren Version ist das *ScanEvent*. Ein *ScanEvent* wird in regelmäßigen Abständen ausgelöst und überträgt immer nur die seit dem letzten Ereignis gefundenen Geräteadressen.

Tabelle 5.3: Inhalt der Ereignisnachrichten

Ereignis	Daten
ScanEvent (1)	A, $t.a$
ScanEvent (2)	B, $t.b$, C, $t.c$
ScanEvent (3)	Wird unterdrückt, da keine neuen Geräte seit dem letzten ScanEvent geantwortet haben
ScanResult	A, B, C, $t.i$
ScanEvent (4)	D, $t.d$
ScanEvent (5)	E, $t.e$
ScanEvent (6)	Wird unterdrückt, da keine neuen Geräte seit dem letzten ScanEvent geantwortet haben
InquiryEvent	A, $t.a$, B, $t.b$, C, $t.c$, D, $t.d$, E, $t.e$

Der zeitliche Abstand zwischen den Ereignissen kann konfiguriert werden ($t.wait$). Außerdem wird kurz vor einem Inquiry ein weiteres Ereignis (*InquiryEvent*) verschickt, welches alle Geräteadressen und Zeiten seit dem letzten *InquiryEvent* enthält. Da die Abwärtskompatibilität beibehalten werden soll, schicken auch BlueScan2-Geräte nach Ablauf der Verzögerungszeit ein *ScanResult*-Ereignis.

Listing 5.13: XML-Schema-Definition für den Konfigurationstyp

```
<xs:complexType name="BlueScanConfigType">
    <xs:sequence>
        <xs:element name="InquiryInterval" type="xs:integer"
            minOccurs="1" maxOccurs="1">
        </xs:element>
        <xs:element name="Block" type="xs:integer"
            minOccurs="0" maxOccurs="1">
        </xs:element>
    </xs:sequence>
    <xs:any minOccurs="0" maxOccurs="unbounded"
        namespace="##other" processContents="lax" />
</xs:complexType>

<xs:element name="BlueScanConfig"
    type="tns:BlueScanConfigType">
</xs:element>
```

Tabelle 5.3 listet die entsprechenden Geräteadressen und Zeiten, die von den einzelnen Ereignisnachrichten übertragen werden. Es wird deutlich, dass ScanEvent-Ereignisse die tatsächlichen Zeiten übermitteln. Haben zwischen zwei ScanEvents keine neuen Geräte geantwortet, werden

Tabelle 5.4: WSDL-Operationen des BlueScan2-Gerätes

Operation	MEP	Parameter
SetConfig	Request-Response	wie in Tabelle 5.1, jedoch zusätzlich optional die Wartezeit zwischen den ScanEvent-Ereignissen (*t.wait*)
ScanResult	Notification	wie in Tabelle 5.1
InquiryEvent	Notification	Out: Liste mit *allen* Adressen seit dem letzten InquiryEvent
ScanEvent	Notification	Out: Liste mit den Adressen seit dem letzten ScanEvent

diese unterdrückt (ScanEvent (3) und (6)). Das ScanResult-Ereignis enthält nur die ersten drei Geräteadressen, D und E gehen verloren.

Listing 5.14: Beispiel für eine Konfiguration

```
<BlueScanConfig>
    <InquiryInterval>30</InquiryInterval>
    <Block>10</Block>
    <Wait>2</Wait>
</BlueScanConfig>
```

BlueScan2 verwendet mit *SetConfig* und *ScanResult* den gleichen Port-Type wie schon BlueScan1. Allerdings soll in der neuen Version mit der Wartezeit zwischen den ScanEvent-Ereignissen in *SetConfig* ein weiterer Konfigurationsparameter angegeben werden können. Anstatt dafür ein neues XML-Schema, einen neuen Port-Type und einen neuen Diensttyp zu deklarieren, wird hier einfach die Erweiterbarkeit von XML-Schema ausgenutzt. Die Schemadefinition (Listing 5.13) ist mit dem *xs:any*-Element erweiterbar gehalten, indem zusätzliche Elemente hinzugefügt werden können. Die Wartezeit kann somit, wie in Listing 5.14 demonstriert wird, eingestellt werden.

Listing 5.15: BlueScan-Device-Typ Version 2

```
<?xml version="1.0" encoding="UTF-8"?>
<t:Relationship xmlns:t="http://www.ws4d.org/templates/">
    <t:Host>
        <t:Type>
            <t:URI>http://www.ws4d.org/bluescan</t:URI>
            <t:localName>BlueScanDevice2.0</t:localName>
        </t:Type>
        <t:Includes>
            <t:URI>http://www.ws4d.org/bluescan</t:URI>
            <t:localName>BlueScanDevice1.0</t:localName>
        </t:Includes>
```

```
12      </t:Host>
13      <t:Hosted>
14          <t:Reference>
15              http://www.ws4d.org/specs/bluescan/bluescan-scan-service-template.xml
16          </t:Reference>
17          <t:ServiceId>
18              http://www.ws4d.org/bluescan/scanservice
19          </t:ServiceId>
20      </t:Hosted>
21      <t:Hosted>
22          <t:Reference>
23              http://www.ws4d.org/specs/bluescan/bluescan-event-service-template.xml
24          </t:Reference>
25          <t:ServiceId>
26              http://www.ws4d.org/bluescan/eventservice
27          </t:ServiceId>
28      </t:Hosted>
29  </t:Relationship>
```

BlueScan2 verwendet zusätzlich zum *BlueScanScanPortType* aus Version 1 den zweiten Port-Type *BlueScanEventPortType*, der die beiden neuen Operationen *InquiryEvent* und *ScanEvent* definiert (Tabelle 5.4). Für diesen Port-Type wird wiederum ein Diensttyp *BlueScanEventService* sowie ein neuer Gerätetyp *BlueScanDevice2.0* (Anhang B) deklariert. Der neue Gerätetyp (Listing 5.15) spezifiziert, wie erwartet, die beiden Diensttypen (Zeilen 13–28) und kennzeichnet den neuen Typ als kompatibel zum BlueScan-Gerät der ersten Version (Zeilen 8–11).

Listing 5.16: Algorithmus des Inquiry-Threads auf dem BlueScan-Gerät 2 (informal)

```
1  inquiry_thread(target_list, t_interval, t_wait){
2      start_time = current_time
3      start_async_inquiry(target_list, t_interval)
4      while(current_time - start_time < t_interval - t_wait){
5          last_list = copy(target_list)
6          sleep(t_wait)
7          diff = diff(target_list, last_list)
8          if(diff.hasMembers){
9              dpws_notify("ScanEvent", diff)
10         }
11     }
12 }
```

Während die Web-Services-Schnittstelle nach außen hin für den BlueScan-Server für beide Versionen gleich erscheint, ändert sich der Algorithmus (die Implementierung) auf dem BlueScan-Gerät. Die Implementierung kann nicht länger einen blockierenden Inquiry-Aufruf verwenden, sondern muss asynchron arbeiten, wie die beiden Algorithmen zeigen.

Der *Inquiry-Thread* (Listing 5.16) startet dazu einen asynchronen Inquiry-Aufruf, der die gesamte Intervallzeit über gilt und eine (zunächst leere) Liste entgegennimmt, die er mit den Adressen und Zeiten der Antworten füllt (Zeile 3). Die anschließende Routine wird solange abgearbeitet, bis das Intervall abgelaufen ist (Zeilen 4–11). In dieser wird jeweils im Zeitabstand der Wartezeit die Ergebnisliste auf neue Einträge überprüft (Zeile 7) und bei neuen Einträgen das ScanEvent ausgelöst (Zeile 9).

Listing 5.17: Algorithmus des BlueScan-Gerätes Version 2 (informal)

```
1  while(true){
2      if(t_interval > 0){
3          start_time = current_time
4          dpws_notify("InquiryEvent", target_list)
5          target_list = new target_list
6          new inquiry_thread(target_list, t_interval, t_wait).start
7          sleep(t_block)
8          dpws_notify("ScanResult", target_list.getAddresses, start_time)
9          sleep(t_interval - t_block)
10     }
11 }
```

Der Hauptalgorithmus (Listing 5.17) sendet vor jedem Inquiry das InquiryEvent mit der letzten Ergebnisliste (Zeile 4). Danach wird eine neue leere Liste initialisiert (Zeile 5) und der oben erläuterte Inquiry-Thread gestartet (Zeile 6). Nach Ablauf der Verzögerungszeit (BlueScan Version 1) wird das ScanResult-Ereignis mit dem aktuellen Stand ausgelöst (Zeile 8).

In Phase II wurde das Szenario um eine neue Version von BlueScan-Geräten erweitert (Abbildung 5.14). Da diese abwärtskompatibel zur ersten Generation sind, braucht der BlueScan-Server zunächst nicht angepasst zu werden. Die Benutzung erfolgt transparent, da die neuen BlueScan-Geräte beim Discovery die gleichen Geräte- und Diensttypen veröffentlichen wie die älteren. Außerdem haben sie die gleiche Schnittstelle.

5.4.3 Phase III: Bewegungsprofile und Anpassung des BlueScan-Servers

Nachdem die zweite BlueScan-Generation erfolgreich getestet wurde und sich bereits im Einsatz befindet, soll der BlueScan-Server angepasst werden, um die verbesserten Funktionalitäten auszunutzen. Im gleichen Zuge soll eine neue Anwendung umgesetzt werden, die ausgewählte Bewegungsprofile aufbereitet und grafisch darstellt.

Für die Anpassung an die BlueScan2-Geräte sind lediglich kleine Änderungen des Initialisierungscodes bzw. der Registrierungsroutine durchzuführen (Listing 5.18). Im Gegensatz zur ersten Version (Listing 5.11) wird nun nach zwei Gerätetypen gesucht, und je nach Gerätetyp werden unterschiedliche Ereignisse abonniert (Zeilen 13–20).

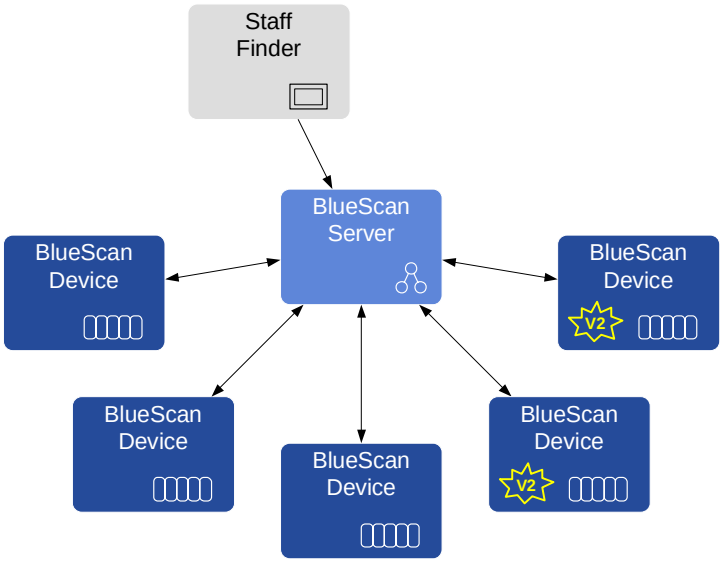

Abbildung 5.14: SOA-Diagramm für BlueScan-Geräte der Versionen 1 und 2 und BlueScan-Server

Außerdem muss ein weiterer Algorithmus die ScanEvent-Ereignisse der BlueScan2-Geräte behandeln (Listing 5.19). Dieser unterscheidet sich von der ScanResult-Ereignisbehandlung (Listing 5.12) lediglich darin, dass jeder Adresse eine Zeit zugeordnet ist (Zeile 4).

Weitere Änderungen sind am BlueScan-Server nicht notwendig. Das Datenbankschema kann beibehalten werden, und auch die Schnittstelle für höhere Dienste bleibt erhalten. Im Szenario waren beispielsweise alle bisher getätigten Änderungen für die *Staff-Finder*-Anwendung nicht sichtbar.

Die neue *Profile-WebApp*-Anwendung verwendet ebenfalls die *Find*-Operation der BlueScan-Server-WSDL. Um Profile über einen langen Zeitraum erstellen zu können, muss sie dazu den *Offset* entsprechend hoch wählen.

Nach Abschluss der ersten drei Szenariophasen können bereits die wichtigsten der in der Einleitung genannten Probleme des BlueTrack-Systems (Abs. 1.3.2) gelöst werden:

- **Lose Kopplung zwischen Sensoren und Server** Die starre Abhängigkeit zwischen Sensoren und Datenbank auf dem Server ist nicht mehr vorhanden. Die Datenbank taucht vor allem gar nicht mehr als Systemkomponente auf, da sie von den SOA-Teilnehmern gekapselt wird. Die Netzwerkadresse des Servers muss den Sensoren zur Entwicklungszeit nicht bekannt sein. Sie ist Teil beim Abonnieren der Ereignisse (zur Laufzeit).
- **Erweiterbarkeit des Systems** Alle Komponenten können, wie demonstriert, unabhängig

Listing 5.18: Algorithmus der Registrierung von BlueScan-Geräten 1 und 2 (informal)

```
namespace = "http://www.ws4d.org/bluescan"
device_type_1 = new qname(namespace, "BlueScanDevice1.0")
device_type_2 = new qname(namespace, "BlueScanDevice2.0")
callback = register
dpws_middleware.listen(device_type, callback)

register(dpws_device){
    if(!bluescan_devices.contain(dpws_device.getUUID)){
        start_time = current_time
        db_insert(new bluescan_device(dpws_device))
        db_insert(new installation("unknown", start_time))
        scan_service = dpws_device.getScanService
        if(dpws_device.hasType(device_type_2)){
            scan_service.subscribe("ScanEvent")
            scan_service.setConfig(30, 10, 2)
        }
        else{
            scan_service.subscribe("ScanResult")
            scan_service.setConfig(30, 10)
        }
    }
}
```

Listing 5.19: Algorithmus beim Empfangen von ScanEvent-Ereignissen (informal)

```
receive_scan_event(dpws_device, targets){
    bluescan_device = find_bluescan_device(dpws_device.getUUID)
    for each target in targets{
        db_insert(new target(
            bluescan_device, target.getAddress, target.getTime))
    }
}
```

voneinander weiterentwickelt werden. Eine manuelle Einrichtung neuer Sensoren ist nicht notwendig, sie werden automatisch erkannt und registriert.

- **Zusätzliche Funktionalitäten** Die Sensoren sind von außen einzeln konfigurierbar. Es lassen sich individuelle (z. B. nach Standort angepasste) Intervalle für das Scannen und Versenden der Ereignisse einstellen.
- **Anwendbarkeit in einem anderen Kontext** Das BlueScan-System lässt sich nun für die Erstellung von Bewegungsprofilen verwenden, unterstützt gleichzeitig jedoch auch eine schnelle Suche von Bluetooth-Geräten. Ein vorbestimmter Anwendungskontext ist dem System entzogen worden. Stattdessen stellen unterschiedliche Dienstnutzer unterschiedliche Anwendungen zur Verfügung (*Staff Finder*, *Profile WebApp*). Die Sensoren lassen sich vor allem auch ohne den BlueScan-Server verwenden, da sie selbst als Dienste im SOA-Netzwerk teilnehmen.

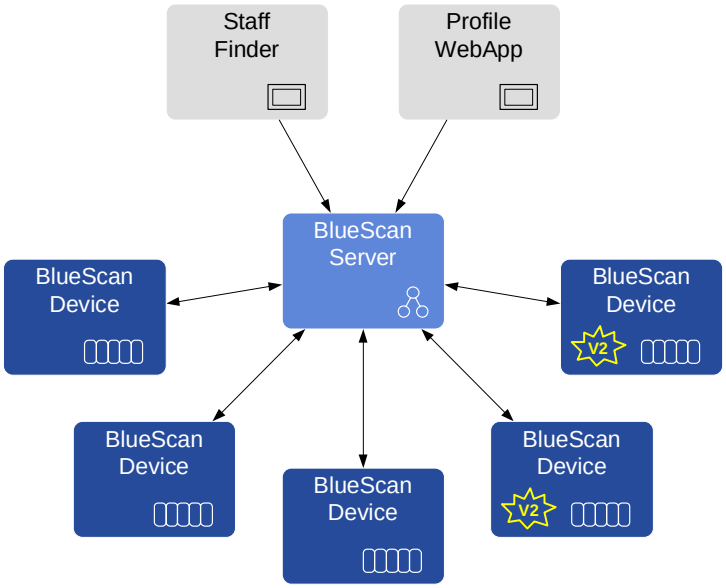

Abbildung 5.15: SOA-Diagramm nach Abschluss der 3. Phase

5.4.4 Phase IV: Discovery-Proxy

Socom hat inzwischen eine beachtliche Anzahl von BlueScan-Geräten im Einsatz, möchte aber weiterhin investieren und das System ausbauen. Aus Skalierungsgründen muss es dazu seine Netzwerkinfrastruktur umstellen. Außerdem sind inzwischen WLAN-basierte und akkubetriebene BlueScan-Geräte erhältlich, die eine einfachere Montage an quasi jedem Ort ermöglichen.

Daraus ergeben sich zwei Probleme: Aufgrund der hohen, typgleichen Geräteanzahl kommt es bei einem *Probe* durch den BlueScan-Server zu einer stark erhöhten Netzwerklast, da alle Geräte auf die Anfrage antworten. Außerdem ist der multicastbasierte Mechanismus von WS-Discovery in der Regel auf lokale Netzwerke begrenzt (abhängig von der Netzwerkkonfiguration), weshalb der BlueScan-Server BlueScan-Geräte, die sich in einem anderen Subnetz befinden, nicht mehr entdecken kann (Abbildung 5.16).

Für beide Probleme kann ein *Discovery-Proxy* eingesetzt werden. WS-Discovery spezifiziert diesen zwar nicht, regelt jedoch das Verhalten von Clients, die die Existenz eines Discovery-Proxies bemerken. Für das erste Problem werden multicastbasierte *Probe*-Nachrichten auf Unicast umgeschaltet, und indem ein Discovery-Proxy auf einem Netzwerkknoten mit mehreren Netzwerkinterfaces implementiert wird, lässt sich gleichzeitig das zweite Problem lösen.

Trotz dieser Restrukturierung können Implementierungsanpassungen der SOA-Teilnehmer gänzlich vermieden werden. Voraussetzung dafür ist eine geeignete DPWS-Middleware auf dem

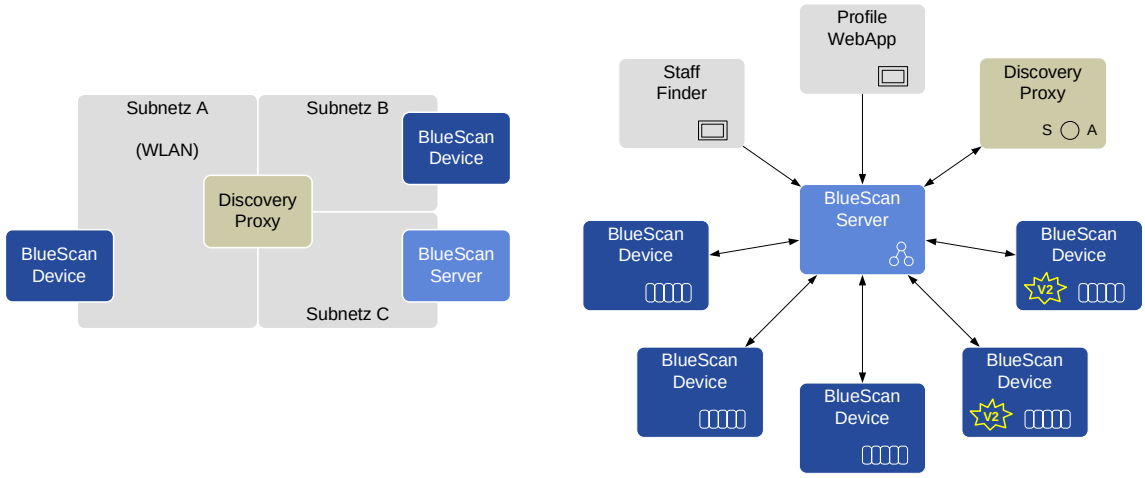

Abbildung 5.16: Discovery über Netzwerkgrenzen hinweg mittels eines Discovery-Proxies

BlueScan-Server sowie ein korrekt arbeitender Discovery-Proxy.

5.4.5 Phase V: Ubisense – Ein neues Lokalisierungssystem

Socom besitzt in seinen Fabrik- und Lagerhallen eine Vielzahl an Fahrzeugen (z. B. Gabelstapler), Paletten, Kisten, Werkzeugen und anderes von den Mitarbeitern gemeinsam genutztes Inventar. Diese ändern ihre Position während der Arbeitsprozesse ständig. Um die Ressourcenplanung (ERP) und auch kurzfristige dynamische Entscheidungsprozesse zu beschleunigen und zu optimieren, soll auch das Inventar lokalisierbar sein. Das hierfür genutzte System muss sehr präzise Messergebnisse liefern. Die Wahl fällt auf Ubisense – ein Lokalisierungssystem für den Innenbereich. Es wird hauptsächlich in der Logistik und in militärischen Anwendungen genutzt. Fest montierte Sensoren bilden die Grundlage für die Messung. Sie sind mit UWB-Empfängern (*Ultra Wide Band*) und RF-Sendern (*Radio Frequency*) ausgestattet. *Tags* sind kleine mobile Systeme, die an die zu messenden Objekte angebracht werden und regelmäßig Signale aussenden. Die Sensoren können anschließend die Positionen mit den beiden Verfahren *Angle Of Arrival* (AoA) und *Time Differences Of Arrival* (TDoA) mit einer Genauigkeit von 30 cm und einer Wahrscheinlichkeit von 95 % ermitteln [138]. Dieses Lokalisierungssystem wird mit einer C++-basierten API ausgeliefert, wodurch die technische Voraussetzung gegeben ist, das System in die vorhandene SOA-Umgebung zu integrieren.

Die WSDL des Ubisense-Gateways verwendet einige spezielle, auf Ubisense ausgerichtete Datentypen (Anhang B). Ein *Point* bildet eine 3D-Koordinate mit den Ordinaten X, Y und Z

Tabelle 5.5: WSDL-Operationen des Ubisense-Gateway-Dienstes

Operation	MEP	Parameter
ReadTags	Request-Response	In: Eine Liste der zu ermittelnden Tags (*TagList*), Out: Die gleiche Liste, jedoch mit den Koordinaten
ReadBox	Request-Response	In: Den auf Tags zu überprüfenden Raum (Quader, *Box*), Out: Liste mit den Tags und Koordinaten, die sich im Raum befinden (*TagList*)
BoxChanged	Notification	Out: Liste mit den Tags und Koordinaten des zu überwachenden Raumes (*TagList*)

ab. Eine *TagList* enthält mehrere *Tag*-Elemente, die jeweils aus *ID* (der interne Identifikator von Ubisense) und optional einer Position *Pos* bestehen. Um Messungen auf bestimmte Räume zu beschränken, kann über das *Box*-Element durch Angabe zweier diagonal gegenüberliegender Punkte (*Point*) ein Quader aufgespannt werden, der den Raum spezifiziert.

Das Ubisense-Gateway bietet drei Operationen an (Tabelle 5.5). Mit *ReadTags* werden die aktuellen Koordinaten bestimmter Tags abgefragt. *ReadBox* ermittelt alle im spezifizierten Raum lokalisierten Tags und gibt deren Positionen zurück. Die *BoxChanged*-Operation ist eine auf WS-Eventing basierende Notification. Mit ihr können Räume überwacht werden. Nach einer entsprechenden *Subscription* schickt das Ubisense-Gateway zunächst die aktuellen Koordinaten der im Raum befindlichen Tags. Danach werden *BoxChanged*-Nachrichten nur gesendet, wenn sich entweder die Koordinaten im überwachten Raum ändern oder aber ein Tag den Raum verlässt bzw. ihn betritt.

Um *BoxChanged* zu abonnieren, muss dem Dienst in irgendeiner Weise der Raum mitgeteilt werden, der überwacht werden soll. Es handelt sich hierbei um ein parametrisiertes Abonnement. Die beiden standardisierten Filter *XPath-Filter* (WS-Eventing) und *Action-Filter* (DPWS) sind an dieser Stelle ungeeignet bzw. nicht verwendbar: Mit *Action-Filter* lassen sich nur Operationen filtern, vor allem jedoch keine Parameter angeben. Mit dem *XPath-Filter* könnte zwar ein Ausdruck formuliert werden, der überprüft, ob und wieviele Tags sich in einem bestimmten Raum befinden, jedoch wird dieser auf eine bereits generierte SOAP-Nachricht angewandt. Über eine solche Nachricht kann jedoch erst dann verfügt werden, wenn aus den Raumparametern die entsprechenden Daten berechnet wurden. Der XPath-Ausdruck wäre danach hinfällig. Außerdem müssten die Parameter in den Ausdruck kodiert werden. Beides ist nicht sinnvoll. Daher muss ein neuer Filter spezifiziert werden. Dieser besitzt eine eigene *Dialect*-URI und nimmt als Inhalt ein *Box*-Element entgegen, welches den zu überwachenden Raum spezifiziert (Listing 5.20).

Listing 5.20: Anwendung des *BoxFilters* beim Senden einer *Subscribe*-Nachricht an das Ubisense-Gateway

```
<soap:Envelope>
    ...
    <soap:Body>
        <wse:Subscribe>
            ...
            <wse:Filter Dialect="http://www.ws4d.org/ubisense/BoxFilter">
                <ubi:Box xmlns:ubi="http://www.ws4d.org/ubisense">
                    <ubi:PointA>
                        <ubi:X>1.2</ubi:X><ubi:Y>0.8</ubi:Y><ubi:Z>0</ubi:Z>
                    </ubi:PointA>
                    <ubi:PointB>
                        <ubi:X>4.3</ubi:X><ubi:Y>1.5</ubi:Y><ubi:Z>1.0</ubi:Z>
                    </ubi:PointB>
                </ubi:Box>
            </wse:Filter>
            ...
        </wse:Subscribe>
    </soap:Body>

</soap:Envelope>
```

Das Ubisense-Gateway kapselt das Ubisense-System, welches ein relativ monolithisches System darstellt. Für Socom wird angenommen, dass eine Anwendung existiert, mit der sich das Inventar lokalisieren lässt (Abbildung 5.17).

5.4.6 Phase VI: Integration heterogener Lokalisierungssysteme

Ubisense und BlueScan sind nun seit einigen Jahren im Einsatz. Socom möchte endlich dazu übergehen, beide Systeme miteinander zu verbinden und transparent abzubilden. Dazu soll ein Lokalisierungsdienst entwickelt werden, dessen Schnittstelle unabhängig von Ubisense oder BlueScan funktioniert, so dass später weitere Lokalisierungssysteme hinzugefügt werden können, ohne die anderen Dienste oder Dienstnutzer, die dann bereits den Lokalisierungsdienst verwenden, zu stören. Der Lokalisierungsdienst soll die Grundlage bilden, um Fragen wie „Welche Mitarbeiter befinden sich gerade in der Nähe von Maschine X oder Fahrzeug Y?“ beantworten zu können. Anwendungen, die auf solchen Informationen basieren (*Ortsbasierte Anwendungen*), ist es in der Regel egal, wie und mit welcher Lokalisierungsinfrastruktur die Daten gewonnen oder aufbereitet werden.

Um den Lokalisierungsdienst unabhängig von spezifischen Lokalisierungssystemen zu machen, muss *Lokalisierung* als Anwendungsdomäne zunächst analysiert werden. Als *Lokalisierungssysteme* werden Infrastrukturen bezeichnet, die entweder die Position von Objekten ermitteln können oder aber Benutzer dazu befähigen, ihre Position selbst zu ermitteln. Daher können Lokalisie-

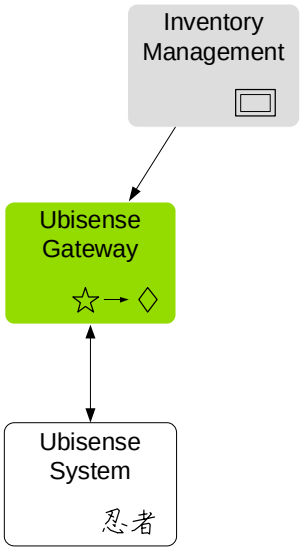

Abbildung 5.17: SOA-Diagramm für die Ubisense-Integration

rungssysteme schon einmal hinsichtlich ihrer Lokalisierungsmethode unterschieden werden. Weiterhin differieren sie bezüglich des Einsatzortes (im Freien oder innerhalb von Gebäuden, lokal oder global), des Kommunikationsmustes (Pull oder Push), ihrer Kosten und ihrer Genauigkeit. Zum Beispiel ist GPS eines der bekanntesten Lokalisierungssysteme, welches seine Dienste weltweit (global) anbieten kann und dessen Genauigkeit im Bereich von 1–20 m liegt. Es ist jedoch nur für die Positionsbestimmung im Freien geeignet.

Bei der *aktiven Lokalisierungsmethode* erfolgt die Positionsbestimmung durch die Infrastruktur selbst. Die Messung wird in der Regel vom Objekt gar nicht bemerkt. BlueScan ist ein Beispiel für die aktive Lokalisierungsmethode. Umgekehrt wird bei der *passiven Lokalisierungsmethode* die Infrastruktur lediglich dazu bereitgestellt, damit ein Benutzer sich selbst lokalisieren kann. Die Messdaten liegen anschließend beim Objekt, nicht wie bei der ersten Methode beim Lokalisierungssystem (z. B. GPS).

Eine *Position* referenziert entweder einen Ort (Namen von Städten, Gebäude, Strassen usw.) oder aber sie steht für einen Punkt, eine Fläche oder einen Körper mit Bezug auf ein bestimmtes *Koordinatensystem*. Ein Koordinatensystem dient der eindeutigen Positionsangabe eines Punktes in einem Raum. Das Konzept der Position deckt damit sowohl *geometrische* (z. B. GPS) als auch *symbolische* (z. B. BlueScan) Positionierungsmodelle ab.

Unter *Objekte* werden alle diejenigen Objekte verstanden, die eine Position besitzen können und die im System von Interesse sind. Objekte können eingeteilt werden in *physikalische* und

logische Objekte. Physikalische Objekte sind real existierende Objekte, die in virtuellen Koordinatensystemen positioniert sind (z. B. Personen, Autos, Mobiltelefone). Logische Objekte sind dagegen lediglich konzeptionelle Elemente (z. B. Dienste), denen eine Position in einem virtuellen Koordinatensystem zugewiesen wird. Daraus ergibt sich, dass Objekte beliebig viele Positionen besitzen können, je nach Anzahl der verwendeten Koordinatensysteme.

Um unabhängig von bestimmten Lokalisierungssystemen zu sein, muss der Lokalisierungsdienst die o. g. Konzepte alle gleichermaßen unterstützen. Die genannten Punkte bilden den abzudeckenden *Entwurfsraum* [118] für den Dienst und sollten nach Möglichkeit nicht durch die (generische) Schnittstelle eingeschränkt werden. Eine Einordnung verschiedener Lokalisierungssysteme findet sich in [139].

Die folgende Aufzählung enthält die Anforderungen bzw. Funktionalitäten, die vom Lokalisierungsdienst angeboten werden sollen. Weitere theoretische Betrachtungen zum Entwurf eines Lokalisierungsdienstes finden sich in [118]. Für die Umsetzung eines konkreten Dienstes siehe auch [117].

- **Lokalisierung** Mit der Lokalisierungsfunktion können Positionen von Objekten ermittelt werden. Damit realisiert die Lokalisierungsfunktion die Abbildung von Objekten auf Positionen.
- **Gebietserfassung** Die Gebietserfassung ist die Umkehrfunktion zur Lokalisierung. Sie nimmt Positionen als Parameter entgegen und antwortet mit den in ihr enthaltenen Objekten.
- **Umgebungsanfrage** Umgebungsanfragen ermitteln Objekte in der Nähe eines Zielobjektes. Eine Umgebungsanfrage ist damit eine etwas anders formulierte Gebietserfassung, bei welcher das Gebiet indirekt durch das Zielobjekt und einen Radius spezifiziert wird.
- **Positionstransformation** Die Positionstransformation stellt Funktionen zum Transformieren zwischen Positionsrepräsentationen verschiedener Positionsmodelle zur Verfügung. Hierzu gehört das Umrechnen von Koordinaten eines Quellkoordinatensystems in ein Zielkoordinatensystem. Ebenso müssen symbolische Bezeichner in Koordinaten bzw. umgekehrt umgerechnet werden können. Über diesen Weg lassen sich auch dann Objektrelationen (z. B. Abstandsermittlungen) berechnen, wenn die Objekte durch unterschiedliche Lokalisierungssysteme gemessen wurden.
- **Ereignisse** Es sollen Ereignisse angeboten werden, um bestimmte Objekte und Gebiete zu überwachen.
- **Auswertung** Für die Erstellung von Bewegungsprofilen und ähnlichen Auswertungen muss der Zugriff auf ältere Daten möglich sein.

Im Folgenden wird die Lokalisierung und Gebietserfassung genauer betrachtet, da sich an diesen beiden die Konzepte des Schnittstellenentwurfes des Lokalisierungsdienstes am besten veranschaulichen lassen. Die entsprechenden WSDL-Dokumente befinden sich im Anhang B.

Um die Schnittstelle des Lokalisierungsdienstes unabhängig von einem bestimmten Objekt- oder Positionsmodell zu gestalten, wird hier das Konzept des *Dialects* eingeführt. Dieses wird bereits von anderen Web-Services-Protokollen verwendet und stellt einen mächtigen Mechanismus dar, um Implementierung und Schnittstelle zu trennen. Ein *Dialect* ist eine Metainformation in Form einer URI und beschreibt, wie die dem Dialekt zugeordneten Daten interpretiert werden müssen. Dialekte werden verwendet, um Sprachen für Objekte, Positionen, Qualifizierer und Qualitätsangaben zu spezifizieren, z. B.:

- *http://www.ws4d.org/ls/dialect/wgs84*
  Das *World Geodetic System 1984* (WGS84) ist ein globales Koordinatensystem, welches für Positionsangaben auf der Erde verwendet werden kann. GPS basiert beispielsweise auf WGS84.
- *http://www.ws4d.org/ls/dialect/gml*
  Die *Geography Markup Language* (GML) stellt einen Standard für die geometrische Beschreibung (*Geometry*) von positionierten Objekten (*Features*) dar.
- *socom:ubisense/storage43*
  Ein Socom-spezifischer Dialekt, der das von Ubisense verwendete Koordinatensystem in der Lagerhalle 43 referenziert.
- *socom:bluescan*
  Ein Socom-spezifischer Dialekt, der für die Ortsbezeichnungen der BlueScan-Geräte verwendet wird.

Qualifier (Qualifizierer) werden in Anfragen verwendet, um Zusatzinformationen für die Parameter zu formulieren. Beispielsweise kann eine Gebietsanfrage eine bestimmte Wahrscheinlichkeit oder Genauigkeit angeben, mit der sich ein Objekt in dem Gebiet befinden soll. Umgekehrt gibt eine *Quality* (Qualität) Zusatzinformationen für Antwortdaten an. Beispielsweise könnte hier die Genauigkeit einer Positionsangabe stehen.

Der Lokalisierungsdienst teilt sich auf in drei Dienste, den beiden Basisdiensten *Object Management* und *Location Management* sowie dem *Location Service* als zusammengesetzten Dienst. Der Object-Management-Dienst verwaltet die unterschiedlichen Repräsentationen der Objekte im System. Er ist vor allem für die Abbildung unterschiedlicher Repräsentationen auf dasselbe Objekt verantwortlich. Beispielsweise kann einem Benutzer sowohl ein Ubisense-Tag als auch ein Bluetooth-Gerät zugeordnet sein. Der Location-Management-Dienst rechnet Koordinaten und

symbolische Ortsbezeichnungen ineinander um. Der Location-Service verwendet diese beiden Dienste zusammen mit den Diensten der Lokalisierungssysteme.

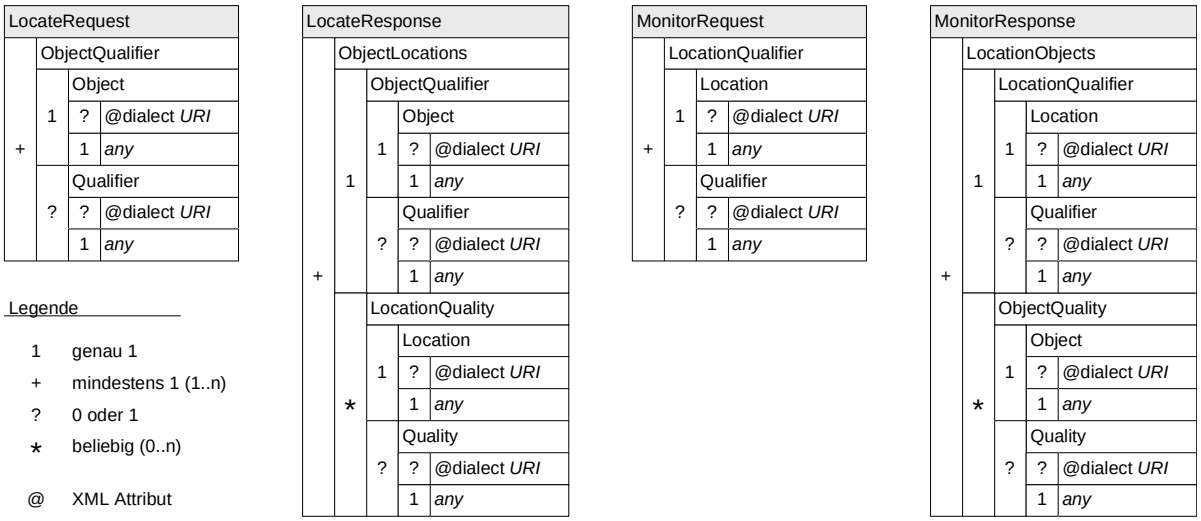

Abbildung 5.18: XML-Struktur der Parameter des Lokalisierungsdienstes

Abbildung 5.18 zeigt den strukturellen Aufbau der Dienstparameter des Lokalisierungsdienstes so, wie er mit XML-Schema im WSDL-Dokument definiert ist. Die drei folgenden Beispiele veranschaulichen die Arbeitsweise des Lokalisierungsdienstes. Die tatsächliche Implementierung ist aus SOA-Sicht verborgen. In beiden Beispielen wird der Einfachheit halber die z-Koordinate des Ubisense-Systems und der Faktor Zeit vernachlässigt.

Ein Dienstnutzer möchte den Mitarbeiter *John* lokalisieren. Dazu ruft er *Locate* vom Location-Service auf. Um den Mitarbeiter (bzw. das Objekt) zu spezifizieren, wird eine entsprechende Objektbeschreibung benötigt. Hier *socom:staff* als Dialekt für Mitarbeiternamen und *John* als Objektqualifizierer (Abbildung 5.19) verwendet.

Der Location-Service reicht die Objektbeschreibung weiter an den Object-Manager und erhält zwei Einträge als Ergebnis der *TranslateObject*-Operation: Eine Ubisense-Tag-ID, die John derzeit zugeordnet ist und einen Login-Namen. Mit dem Login-Namen kann der Location-Service (derzeit noch) nichts anfangen, weshalb er ihn ignoriert. Es wird hier angenommen, dass Socom gerade die Login-Informationen ihrer Arbeitsplätze als zusätzliches Lokalisierungssystem benutzt, diese Umsetzung im Object-Manager bereits abgeschlossen ist, im Location-Manager jedoch noch nicht begonnen wurde. Mit Hilfe des Dialekts kann der Location-Service den ersten Eintrag Ubisense zuordnen und ruft in der Folge die *ReadTag*-Operation des Ubisense-Gateways auf. Diese gibt einen Punkt in Form von Koordinaten zurück. Die Koordinaten beziehen sich auf

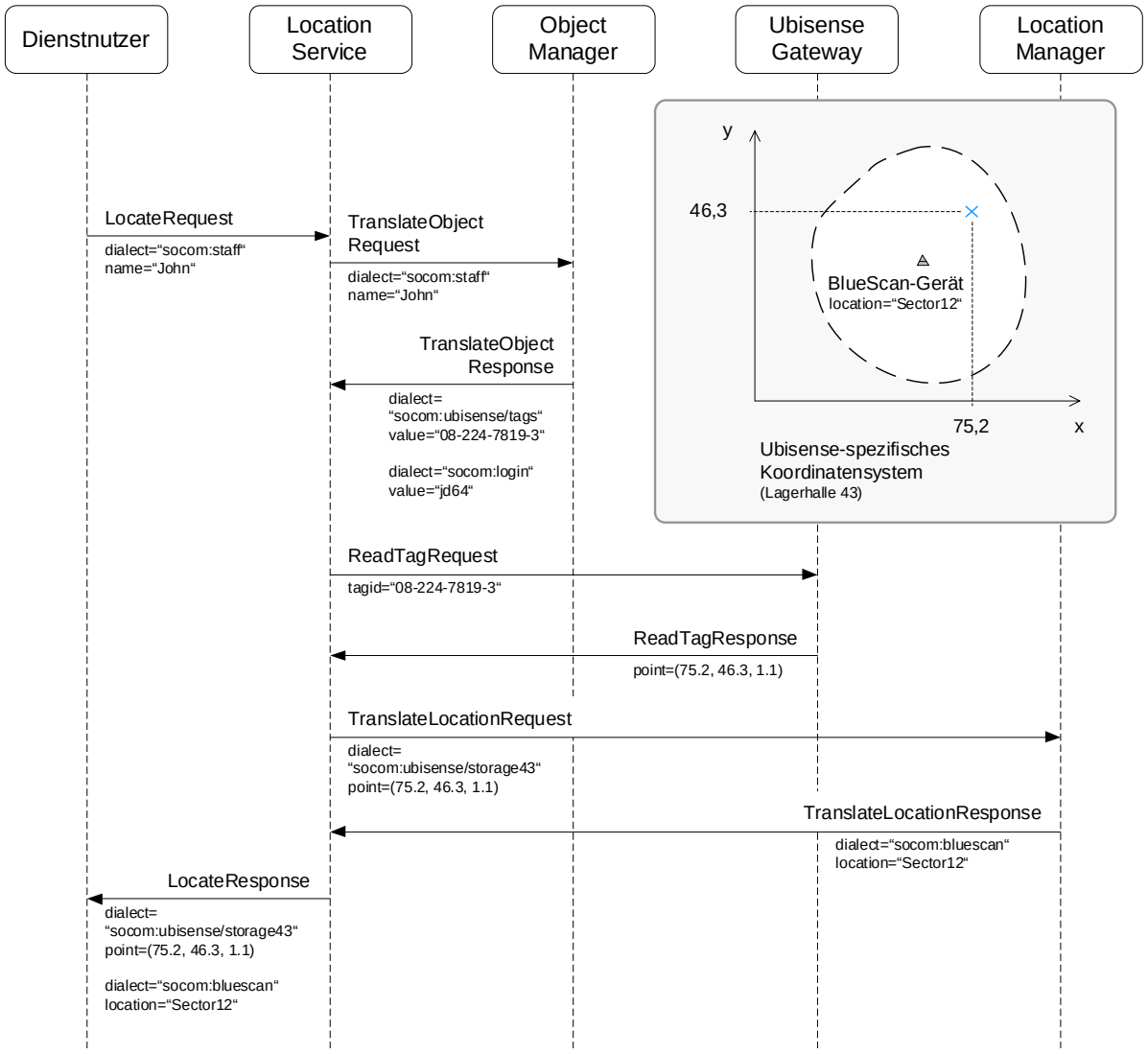

Abbildung 5.19: Dienst-Sequenzdiagramm zur Lokalisierung eines Mitarbeiters

das Ubisense-spezifische Koordinatensystem, welches bei der Installation des Systems kallibriert wurde. Damit ist der Mitarbeiter bereits lokalisiert. Um die ermittelte Position auch noch in andere Positionsmodelle umzurechnen, ruft der Location-Service die *TranslateLocation*-Operation im Location-Manager auf. Wie im Sequenzdiagramm ersichtlich, verwendet der Location-Service hierfür *socom:ubisense/storage43* als Dialekt für die ermittelte Position. Das Konzept der Dialekte wurde durch den Lokalisierungsdienst (bestehend aus den drei Diensten) eingeführt. Die Abbildung von einem Dialekt auf ein Koordinatensystem bzw. ein anderes Positionierungsmodell muss daher durch den Lokalisierungsdienst erfolgen. Es wird in diesem Beispiel angenommen, dass nur *ein* Ubisense-Gateway (in der Lagerhalle 43) verwendet wird. Sind mehr als eines solcher Systeme im Einsatz, muss der Location-Service alle Systeme nacheinander anfragen (*ReadTag*). Die Abbildung von Dialekt auf Koordinatensystem kann über die eindeutige Geräte-UUID erfolgen.

Ebenfalls ersichtlich ist in Abbildung 5.19, dass sich der gemessene Punkt innerhalb der Reichweite des BlueScan-Gerätes befindet, welches dem Ort *Sector12* zugeordnet ist. Es wird hier angenommen, dass der Einflussbereich des BlueScan-Gerätes durch eine Messwertreihe im Voraus erfasst wurde. Die entsprechenden Informationen verwaltet der Location-Manager. An dieser Stelle wird eine weitere Eigenschaft von Serviceorientierung deutlich: Dienste besitzen nicht per Definition eine Eins-zu-Eins-Beziehung zu einer Softwarekomponente oder einer anderen Architekturkomponente in der darunterliegenden Softwarearchitektur. Im Beispiel wird das daran deutlich, dass der Location-Service den Ort *Sector12* zurückgibt, diese Information aber vom BlueScan-Server bzw. dessen Datenbank verwaltet wird, siehe auch Abbildung 5.11. Ob der Location-Manager diese Daten selbst über die SOA-Infrastruktur[7] oder aber über einen einfachen Datenbankaufruf bezieht, ist auf Dienstebene nicht ersichtlich. *LocateResponse* liefert schließlich zwei Positionen zurück.

Das zweite Beispiel demonstriert die Verwendung von Qualifizierern bzw. Qualitätsangaben (Abbildung 5.20). Die Lokalisierung soll dieses Mal für die Mitarbeiterin *Anne* erfolgen. Zusätzlich spezifiziert die Anfrage eine einzuhaltende Wahrscheinlichkeit p_{incl} von 100% sowie einer erforderlichen Mindestqualität p_{qual} von 50%. Innerhalb des Dialektes *socom:simple* drückt p_{incl} die Wahrscheinlichkeit aus, mit der sich ein lokalisiertes Objekt an einem Ort befindet. Die Qualität gibt die Güte einer Lokalisierung an. Die eigentliche Interpretation dieser Angaben ist von einem zum anderen Lokalisierungssystem bzw. Positionsmodell unterschiedlich.

Der Object-Manager gibt für das Objekt *Anne* eine Bluetooth-Geräteadresse zurück, die ihr zugeordnet ist. Der Location-Service kann aufgrund des Dialektes *socom:bluescan/btdevid* das Objekt dem BlueScan-Lokalisierungssystem zuordnen und in der Folge die *Find*-Operation auf-

[7] oft auch: *Enterprise Service Bus* (ESB)

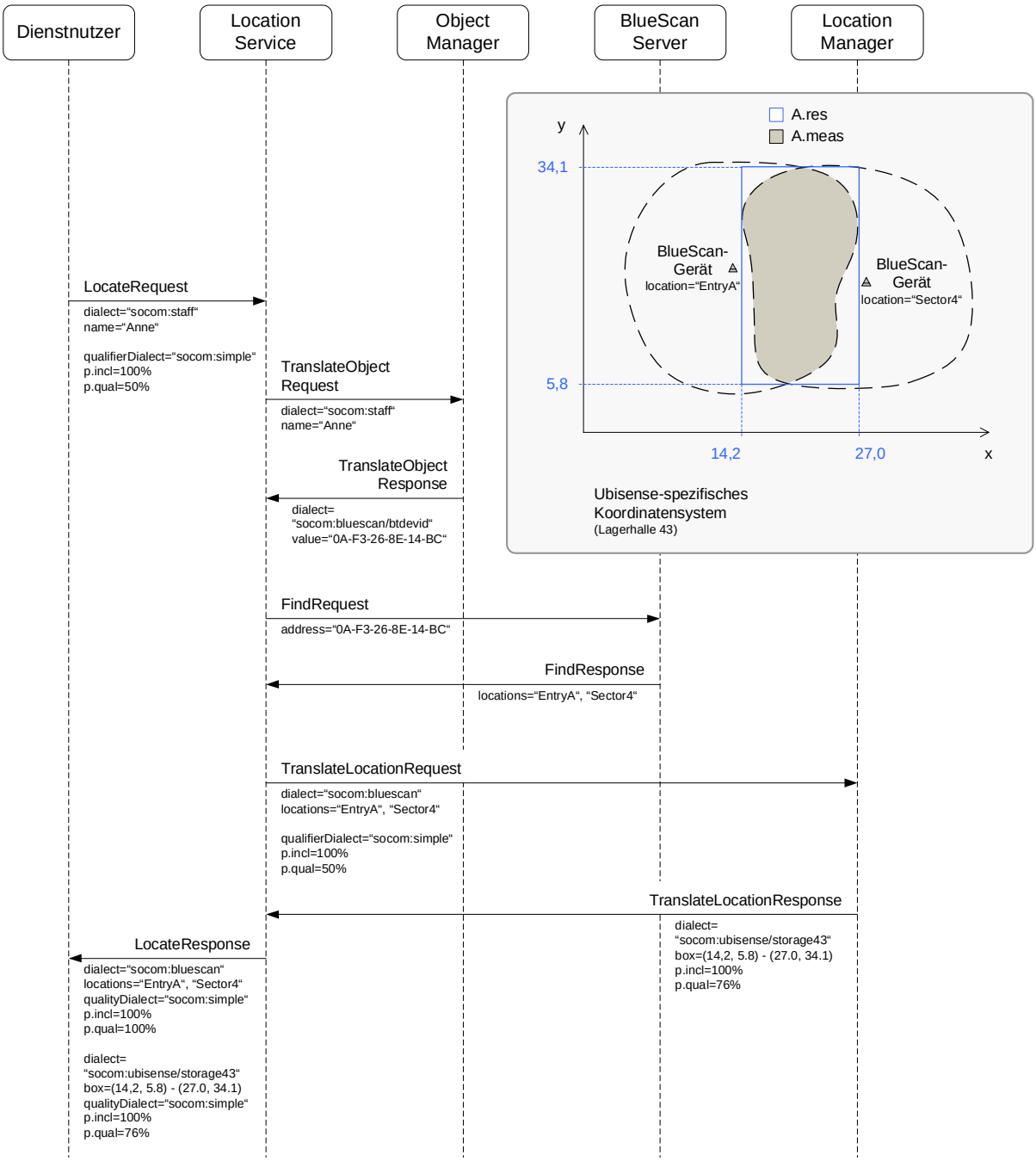

Abbildung 5.20: Dienst-Sequenzdiagramm zur Lokalisierung eines Mitarbeiters mit Angabe eines Qualifizierers ($p_{incl} = 100\%$)

rufen. Die Adresse wurde von zwei BlueScan-Geräten zeitgleich entdeckt, so dass der BlueScan-Server zwei Orte *EntryA* und *Sector4* zurückgibt. Die Messung muss demnach im überlappenden Bereich (Fläche A_{meas}) erfolgt sein. Diese Fläche stellt bereits das erste Ergebnis der Lokalisierungsanfrage dar. Um weitere Positionsangaben zu erhalten, wird das Ergebnis an den Location-Manager überreicht. Dieser interpretiert die beiden Orte korrekt als Überlappung und gibt aufgrund der geforderten Wahrscheinlichkeit von 100% das Rechteck zurück, welches die überlappende Fläche umschreibt. Es wird hier wieder davon ausgegangen, dass zuvor eine entsprechende Kallibrierung bzw. Einmessung stattgefunden hat. Die Qualität beträgt hier 76%[8].

Für Ubisense-spezifische Koordinatensysteme interpretiert und berechnet der Location-Service die Wahrscheinlichkeiten wie folgt: Die Wahrscheinlichkeit p_{incl}, dass sich das Objekt in der vom Location-Manager zurückgegebenen Fläche A_{res} befindet, entspricht dem Verhältnis aus der Überlappungsfläche A_o (Durchschnitt der Flächen A_{meas} und A_{res}) zur tatsächlich gemessenen Fläche A_{meas}. Im vorliegenden Fall sind die Überlappungsfläche A_o und die gemessene Fläche A_{meas} gleich groß, so dass p_{incl} 1 ist. Die Qualität dagegen berechnet sich aus p_{incl} multipliziert mit dem Verhältnis der Überlappungsfläche A_o zur zurückgegebenen Fläche A_{res}:

$$p_{incl} = \frac{A_o}{A_{meas}} = \frac{A_{meas} \cap A_{res}}{A_{meas}}$$

$$p_{qual} = p_{incl}\frac{A_o}{A_{res}} = p_{incl}\frac{A_{meas} \cap A_{res}}{A_{res}} = \frac{(A_{meas} \cap A_{res})^2}{A_{meas}A_{res}}$$

Das dritte Beispiel ist ähnlich dem zweiten mit dem Unterschied, dass hier eine möglichst hohe Qualität der Ergebnisse gefordert ist. Der Location-Manager gibt daher ein Rechteck zurück, welches sich der tatsächlichen Fläche gut nähert. Die Qualität des Ergebnisses ist mit 81% zwar etwas höher als im zweiten Beispiel, die Wahrscheinlichkeit, dass sich *Anne* tatsächlich in diesem Rechteck befindet, ist dagegen nur noch 89% groß[9].

Nach Abschluß der sechsten Phase ergänzen drei weitere Dienste das SOA-Diagramm: Der *Location-Manager* und der *Object-Manager* sind Basisdienste, verwenden selbst also keine anderen Dienste. Der Location-Manager kapselt die Lokalisierungssysteme und die beiden genannten Basisdienste, womit er aus SOA-Sicht eine Fassade darstellt. Zudem bietet er neue Dienste wie Umgebungssuche an (Abbildung 5.22).

[8] für folgende Annahme: $A_{meas} = 275$
[9] für folgende Annahmen: $A_{meas} = 275$, $A_o = 245$

Abbildung 5.21: Dienst-Sequenzdiagramm zur Lokalisierung eines Mitarbeiters mit Angabe eines Qualifizierers ($p_{qual} = best\ quality$) (Ausschnitt)

5.4.7 Phase VII+: Zusammenfassung und Ausblick

Das Szenario hat gezeigt, wie das Devices Profile for Web Services und das Paradigma der serviceorientierten Architekturen beim Aufbau einer auf Langfristigkeit und Wiederverwendbarkeit ausgelegten IT-Infrastruktur helfen können. Als Anwendungsdomäne wurde eine Lokalisierungsplattform gewählt, da hier u. a. auch Geräte zum Einsatz kommen. In jeder Phase wurden unterschiedliche Konzepte demonstriert:

Phase I Deklaration von Geräte- und Diensttypen
Verwendung der Geräte-UUID
Dienst als Fassade (BlueScan-Server)
Dynamische Registrierung von Geräten eines bestimmten Typs (auf dem BlueScan-Server)
Verwendung von Abonnements (WS-Eventing)

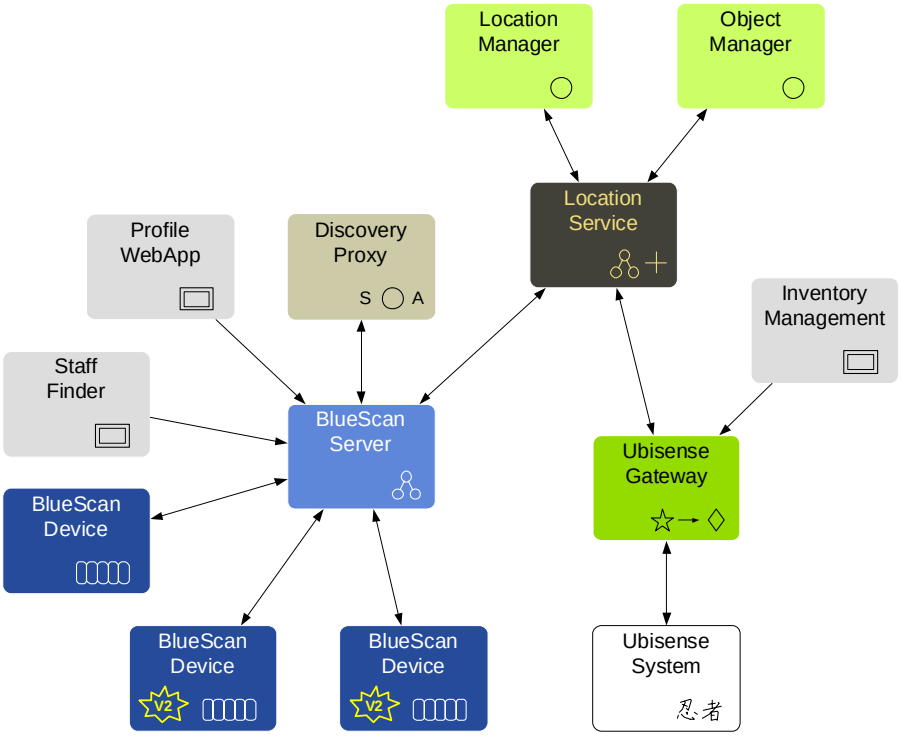

Abbildung 5.22: SOA-Diagramm für die Integration heterogener Lokalisierungssysteme

Phase II	Kompatibilität durch Deklaration von abwärtskompatiblen Geräte- und Diensttypen Kompatibilität durch Ausnutzung von XML-Schema
Phase III	Iterative Weiterentwicklung einer Dienstimplementierung (BlueScan-Server) durch Verwendung verbesserter Dienste (neue Versionen), ohne die eigene Schnittstelle zu modifizieren
Phase IV	Einsatz eines Discovery-Proxies
Phase V	Deklaration eines eigenen WS-Eventing-Filters
Phase VI	Reduktion von vorhandener Komplexität sowie Hinzufügen neuer Funktionalität durch Einsetzen einer Fassade (Location-Service) Langfristige Wiederverwendbarkeit einer Schnittstelle durch Entwurf grobkörniger Operationen und Verwendung des generischen Dialektkonzeptes

Weiterhin ist die SOA nach jeder Phase intakt geblieben. Anwendungen und Dienste, die gleichzeitig Dienstnutzer waren, konnten jederzeit weiter funktionieren.

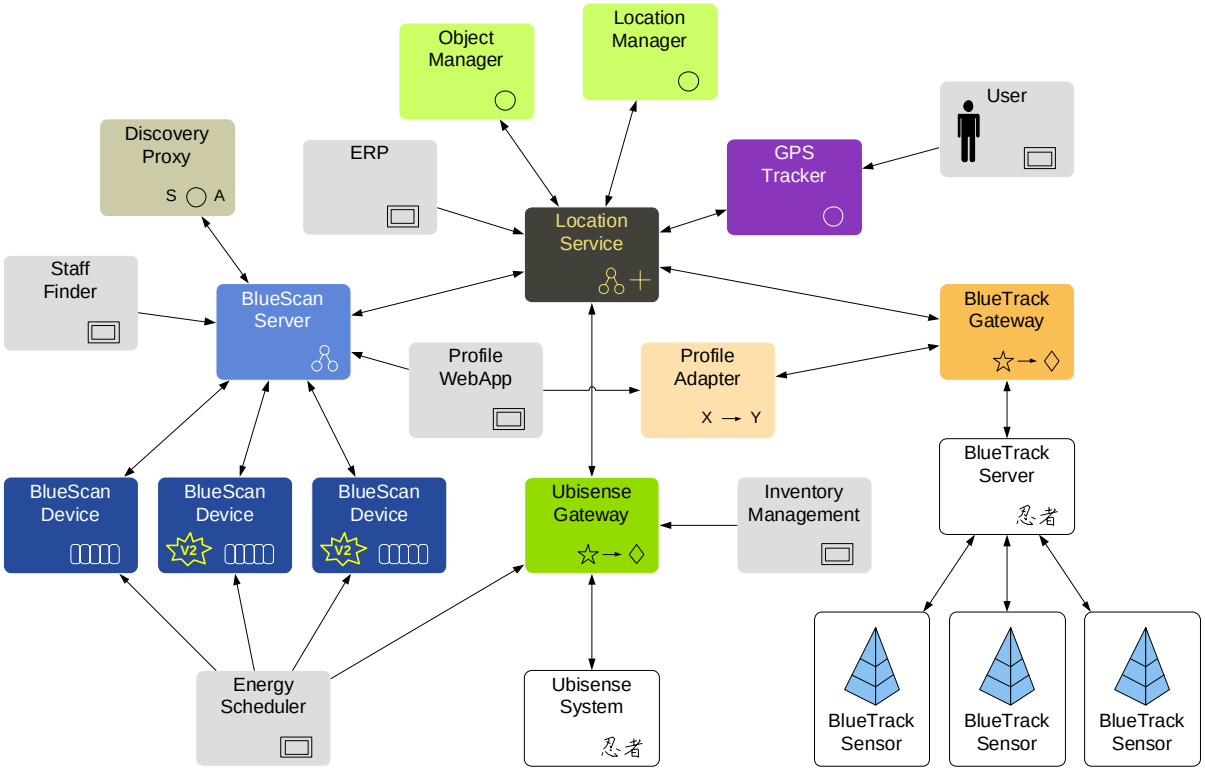

Abbildung 5.23: Einfache Erweiterbarkeit durch Integration neuer Dienste

Abbildung 5.23 zeigt eine mögliche Fortführung der SOA. Mit dem *BlueTrack-Gateway* wird der Lokalisierungsplattform ein weiteres bereits existierendes Lokalisierungssystem hinzugefügt. Da dieses ursprünglich die Inspirationsquelle für BlueScan war, basiert es auf dem gleichen Prinzip und ist somit potenziell als Datenquelle für die *Profile-WebApp* aus Abschnitt 5.4.3 geeignet. Dazu wird ein *Profile-Adapter* dem *BlueTrack-Gateway* vorgesetzt, der lediglich die erwarteten Signaturen für die Anwendung zur Verfügung stellt. Benutzer, die über einen GPS-Receiver verfügen, können regelmäßig ihre GPS-Position am *GPS-Tracker* aktualisieren. Dieser Basisdienst stellt seine Daten dann dem Location-Service zur Verfügung. Mit diesem Ansatz können auch Lokalisierungssysteme mit passiver Lokalisierungsmethode in die Plattform integriert werden. Eine *Enterprise Resource Planning* (ERP) Anwendung nutzt den Location-Service, um in Echtzeit Mitarbeiter zu finden, die sich in der Nähe bestimmter logistischer Güter befinden. Und schließlich werden die BlueScan-Geräte und das Ubisense-Gateway um einen neuen Diensttyp

PowerService (siehe auch Abschnitt 5.2.3) ergänzt, über den die Geräte in einen Ruhemodus versetzt werden können, in welchem sie nicht mehr aktiv messen. Der *Energy-Scheduler* erkennt ausschließlich Geräte vom Typ *PowerService*. Mit ihm lassen sich Pläne erstellen und ausführen, die zeitgenau die Geräte schlafen legen und wieder in Bereitschaft versetzen.

5.5 Implementierungen

Der Axis2-basierte DPWS-Stack ist nicht für eingebettete Geräte konzipiert, sondern v. a. für komplexere Dienste und Dienstnutzer auf Enterprise-Ebene geeignet (BlueScan-Server, Location-Service usw.). Daher wurde für die praktische Umsetzung noch ein weiterer Stack eingesetzt, der Ressourcen-arme Geräte adressiert. Der ebenfalls an der Universität Rostock entwickelte DPWS-Stack (*WS4D-gSOAP*) [97] ist in der Programmiersprache C realisiert und setzt auf gSOAP [133] auf. Durch die gleichzeitige Entwicklung und den Einsatz dieser beiden Stacks konnte die Spezifikation intensiver evaluiert werden. Außerdem wurde in diesem Zuge die Interoperabilität zwischen beiden Middleware-Lösungen erhöht. Einige Probleme und Lösungen, die bei der gemeinsamen Evaluierung identifiziert wurden, sind in [94] beschrieben.

5.5.1 BlueScan-Geräte

Die BlueScan-Geräte wurden als eingebettete Systeme auf einem FOX Board realisiert. FOX Boards sind sehr kompakte (66 x 72 mm) Hardwareplattformen mit einer 100 MHz RISC CPU, einem Ethernet und zwei USB-Anschlüssen. Sie sind standardmäßig mit einem Linux-Kernel, einem Web-Server, FTP, TELNET usw. ausgestattet. Um FOX Boards Bluetooth-fähig zu machen, wurden Bluetooth-USB-Sticks verwendet. Als Bluetooth-Software kam der C-basierte BlueZ-Stack zum Einsatz. BlueZ [14] ist eine komplette Bluetooth-Implementierung für Linux-Systeme.

Die DPWS-Anbindung wurde über WS4D-gSOAP realisiert. Dieser verwendet den Plug-in-Mechanismus von gSOAP, um die Erweiterungen für WS-Addressing, WS-Discovery, WS-MetadataExchange, WS-Transfer und WS-Eventing zu integrieren. WS4D-gSOAP benötigt als Ausgangsdaten das BlueScan-WSDL-Dokument sowie die Geräte-Metadaten. Der modifizierte Codegenerator erzeugt Coderümpfe für die Dienste und das Gerät.

5.5.2 BlueScan-Server

Der BlueScan-Server ist hinsichtlich der BlueScan-Geräte ein Dienstnutzer, gleichzeitig fungiert er jedoch als Dienst für den Location-Service. Hierfür kam der Axis2-DPWS-Stack zum Einsatz,

da sich mit diesem sehr einfach beide Rollen umsetzen lassen. Außerdem wurde für die Speicherung der gesammelten Daten eine Datenbank benötigt. Dazu wurde das im Abschnitt 5.4.1 vorgestellte Entity-Relationship-Modell für eine MySQL-Datenbank verwendet.

5.5.3 Ubisense-Gateway

Für die Dienstimplementierung des Ubisense-Lokalisierungssystems kam ebenfalls der WS4D-gSOAP-Stack zum Einsatz. Die Anbindung an Ubisense erfolgte über die Ubisense-API. Da Dienstimplementierung und WS4D-gSOAP in der Programmiersprache C geschrieben sind, die Ubisense-API jedoch in C++, mussten die Funktionen der Bibliothek vom C++-Compiler so erstellt werden, dass diese auch in einem C-Programm verwendet werden können. Hier kam die dafür vorgesehene Speicherklasse *extern „C“* zum Einsatz.

Für die Implementierung der *ReadBox*-Operation müssen jedesmal alle Tags abgefragt werden, die in der entsprechenden Ubisense-Zelle registriert sind. Erst danach kann ermittelt werden, ob der Tag im von *ReadBox* spezifizierten Gebiet liegt. Der Umweg ist notwendig, da die Ubisense-API keine entsprechende Funktionalität anbietet. Die Berechnung erfolgt jedoch sehr zügig, da lediglich Koordinaten miteinander verglichen werden müssen.

5.5.4 Discovery-Proxy

Für die Umsetzung eines Discovery-Proxies wurde das eigene Discovery-Modul verwendet, und es war ein weiterer Eingriff notwendig: Discovery-Proxies müssen auf über Multicast verschickte Probe- und Resolve-Nachrichten sofort, also ohne die sonst vorgeschriebene Wartezeit, mit einem Unicast-Hello-Paket antworten. Für dieses Verhalten wurde unter Ausnutzung des Delegation-Mechanismus (Abs. 5.1.3) eine eigene Klasse implementiert. Diese speichert außerdem *alle* Dienste bzw. Geräte in einem Cache, um auf über Unicast verschickte Probe- und Resolve-Anfragen antworten zu können.

5.5.5 Location-Service, Location-Manager und Object-Manager

Der Location-Service wurde ebenfalls mit dem Axis2-DPWS-Stack umgesetzt. Die beiden Dienste Location-Manager und Object-Manager wurden nicht separat implementiert, sondern sind im Prototyp im Location-Service mit enthalten. Diese können, wie zuvor dargestellt, auch als eigene Dienste ausgelagert werden. Dadurch ändert sich jedoch nicht das Verhalten des Location-Services. Die praktische Evaluierung beschränkte sich weiterhin auf zwei Räume; beide waren mit BlueScan, einer mit Ubisense ausgestattet. Die Positionstransformation gibt daher für das

BlueScan-Gerät im Ubisense-Raum immer die gesamte Raumabmaße im Ubisense-spezifischen Koordinatensystem zurück bzw. umgekehrt.

5.6 Zusammenfassung

Das Kapitel zeigte, wie das Devices Profile for Web Services verwendet werden kann, um eine serviceorientierte Architektur aufzubauen. Dazu wurde zunächst die eigene, auf Java basierende DPWS-Implementierung für das Apache-Axis2-Projekt vorgestellt, welche innerhalb der WS4D-Initiative entwickelt wurde. Die Umsetzung erfolgte dabei modular, indem die einzelnen Web-Services-Spezifikationen, die für DPWS benötigt werden, als abgeschlossene Axis2-Module implementiert wurden. Dadurch können zukünftig auch andere Web-Services-Profile oder andere Anwendungen diese Module benutzen.

Beim Aufbau einer Lokalisierungsplattform wurde gezeigt, wie geeigneter Dienstentwurf und die Ausnutzung des SOA-Paradigmas beim Aufbau einer auf Langfristigkeit und Wiederverwendbarkeit ausgelegten IT-Infrastruktur helfen können. Unter der Verwendung des Devices Profile for Web Services wurden Möglichkeiten erörtert, die SOA auszubauen und gleichzeitig intakt zu lassen, so dass Anwendungen und Dienste jederzeit weiter funktionieren konnten.

Durch BlueScan konnten alle im Abschnitt 1.3.2 identifizierten Nachteile des BlueTrack-Systems behoben werden. Beim Schnittstellenentwurf des Lokalisierungsdienstes wurde das Konzept der Dialekte verwendet. Dadurch lassen sich zukünftig neue Lokalisierungssysteme integrieren und nutzen, ohne bestehende Anwendungen, beispielsweise ortsabhängige Dienste, neu implementieren zu müssen. Die Integration heterogener Lokalisierungssysteme wurde exemplarisch mit Ubisense und BlueScan dargestellt. Durch die durchgängige Abbildung auf das Konzept der Dienste können spezifische Lokalisierungsdienste in der gleichen SOA auch weiterhin genutzt werden.

Weiterhin wurde eine Möglichkeit vorgestellt, Geräte- und Diensttypen für DPWS formal zu definieren [140, 141], da ein entsprechender Mechanismus bisher fehlte. Die Vorteile des dargestellten Ansatzes sind die Verwendung der XML-Notation, wodurch Codegenerierung ermöglicht wird, Unterstützung von Versionierung und die Abbildung der Typen auf QNames, wodurch sich diese direkt im Discovery verwenden lassen.

Schließlich wurden Details der prototypischen Anbindung der Lokalisierungssysteme vorgestellt, die in der Praxis erfolgreich getestet werden konnten.

6 Die Web-oriented Device Architecture

In den vorangegangenen Kapiteln wurde Universal Plug and Play (UPnP) (Abs. 2.7) und das Devices Profile for Web Services (DPWS) beschrieben (Abs. 2.8). Es wurde demonstriert, wie mit DPWS eine serviceorientierte Architektur umgesetzt werden kann (Abs. 5.4), und es wurde die eigene DPWS-Implementierung vorgestellt (Abs. 5.1).

In diesem Kapitel soll nun mit der *Web-oriented Device Architecture* (WODA) konzeptionell ein eigener Ansatz vorgestellt werden, der eine Alternative zu UPnP und DPWS darstellt. Die Suche nach einer solchen Alternative ist u. a. aus folgenden Gründen gerechtfertigt:

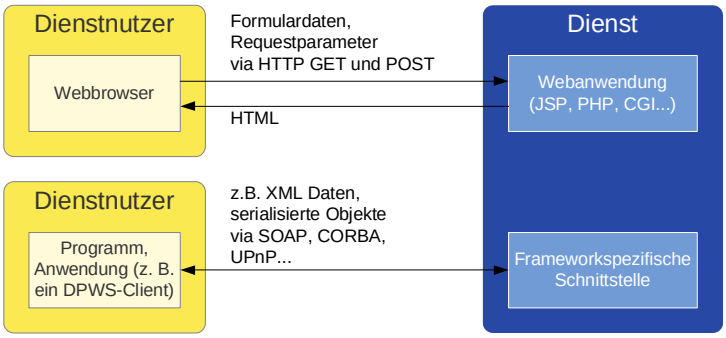

Abbildung 6.1: UPnP und DPWS: Für Geräte mit browserbasierter Steuerung sind zwei unterschiedliche Schnittstellen notwendig

- UPnP verwendet mit dem Simple Service Discovery Protocol (SSDP) und der Generic Event Notification Architecture (GENA) zwei proprietäre Protokolle, die ausschließlich in UPnP zum Einsatz kommen. SSDP hat nie den Status eines Standards erreicht und skaliert sehr schlecht [142], da es ausschließlich Multicast-Discovery verwendet und kein Konzept für eine optionale, zentrale Komponente vorsieht, die für Unicast-basiertes Discovery verwendet werden könnte.
- Der Ereignismechanismus ist sehr grob entworfen, da ein Control Point stets *alle* Zustandsvariablen eines UPnP-Dienstes abonnieren muss, auch dann, wenn er nur an *einer* interessiert ist. Die Notification-Nachrichten der anderen Zustandsvariablen erhält der Control Point dennoch. Die Möglichkeiten eines Control Points reduzieren sich damit auf die beiden

Alternativen, entweder alle Ereignisse eines Dienstes oder sie gar nicht zu abonnieren.

- DPWS verwendet eine Vielzahl von Web-Services-Protokollen, die derzeit immer noch großen Änderungen und Anpassungen unterliegen und von einer Konsolidierung weit entfernt sind. DPWS wird daher zukünftig noch weiter angepasst werden.
- Auch in Ad-hoc-Vernetzungen ist die Konfiguration und die Benutzung von Geräten durch einen Benutzer nach wie vor wichtig. In der Regel geschieht dieses mit einem herkömmlichen Webbrowser. Dazu bieten die Geräte sowohl in UPnP als auch in DPWS eine spezielle URL für die Präsentation an. Für Geräte bzw. Dienste bedeutet dieser Ansatz jedoch, dass sie neben der frameworkspezifischen Schnittstelle eine zweite, zusätzliche Schnittstelle für eine Webanwendung implementieren müssen, die mit einem Webbrowser zusammenarbeiten kann (Abbildung 6.1).

WODA hebt die genannten Einschränkungen unter Beibehaltung der universellen Anwendbarkeit auf, indem es mit Zeroconf (DNS-SD) auf ein etabliertes Discoveryprotokoll setzt, einen Ereignismechanismus zur Verfügung stellt, der mehr Optionen für einen Dienstnutzer erlaubt, keine Web-Services-Protokolle verwendet und mit nur einer Schnittstelle für beide Dienstnutzer (Maschine, Webbrowser) auskommt.

6.1 Einführung in WODA

6.1.1 Architektonische Grundlage

WODA verwendet den Representational State Transfer (REST) als architektonische Grundlage und HTTP als Anwendungsprotokoll, mit welchem sich eine REST-Architektur umsetzen lässt. REST wurde bereits im Abschnitt 2.4 beschrieben. Auf die wichtigsten Konzepte soll hier jedoch noch einmal kurz eingegangen werden.

REST-Komponenten besitzen alle die gleiche architektonische Schnittstelle, die durch die vier Bedingungen Identifizierung von Ressourcen, Manipulation von Ressourcen durch Repräsentationen, selbstbeschreibende Nachrichten sowie Hypermedia als Zustandsmaschine der Anwendung gekennzeichnet ist, siehe auch Abschnitt 2.4. Ressourcen sind dabei Konzepte oder Informationen, die eine feste Semantik besitzen und untereinander verlinkt sind (Hypermedia).

Die Verwendung von REST bedeutet für WODA, dass alle Geräte bzw. Dienste als Mengen von Ressourcen modelliert werden, die zusammen den Zustand des Dienstes darstellen. An diese können dann mittels HTTP semantisch fest definierte Methoden gesendet werden. Beispiele für derartige Ressourcen sind die aktuelle Temperatur, die ein Sensor ermittelt, die Umdrehungszahl eines Motors, der Energiestatus eines Akkus, die Lokalisierungsdaten eines Benutzers oder

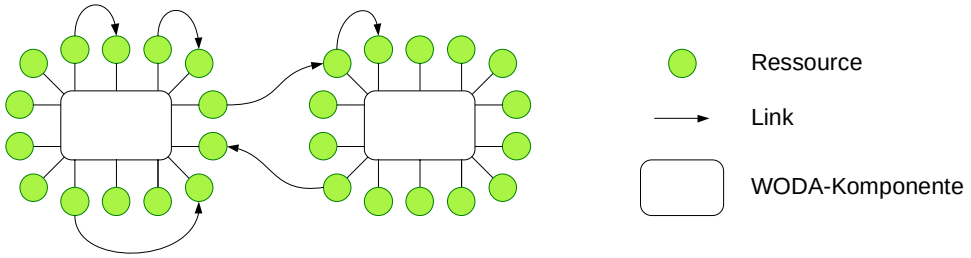

Abbildung 6.2: WODA-Komponenten, Ressourcen und Links

die Liste der MP3-Dateien eines Media-Servers. Alle diese Konzepte können durch Ressourcen abgebildet werden. Die „Größe“ einer Ressource ist unbegrenzt. Es kann sich dabei genauso um einen Zahlwert (z. B. Temperatur) wie um eine komplette Datenbank handeln. Wie später demonstriert, lassen sich mit Ressourcen nicht nur Daten abbilden, sondern auch Konzepte wie Diensttypen oder Abonnements für Ereignisse. Der Vorteil dieses Ansatzes in WODA ist die Gleichbehandlung aller Ressourcen auf Anwendungsebene. Dadurch können gleiche Werkzeuge (z. B. Firewalls, Caches, Webbrowser usw.) unabhängig von konkreten Diensten oder Ressourcen eingesetzt werden.

Abbildung 6.2 zeigt schematisch zwei WODA-Komponenten mit ihren Ressourcen. Diese sind untereinander verlinkt. Durch das Linkkonzept muss ein Dienstnutzer nicht alle Ressourcen eines Dienstes kennen, sondern kann diese zur Laufzeit über die Repräsentationen der Ressourcen erfahren. Beispielsweise wird die Repräsentation der MP3-Dateiliste des erwähnten Media-Servers Links auf die einzelnen MP3-Dateien enthalten. Diese stellen wiederum eigenständige Ressourcen dar, auf die ebenfalls mit dem HTTP-Protokoll zugegriffen werden kann.

Die Verwendung von Ressourcen als Basiskonzept bedeutet für WODA, dass Mechanismen benötigt werden, um Ressourcen zu entdecken, zu beschreiben, zu abonnieren und zu verwenden.

6.1.2 Die sechs Phasen in WODA

Die Integration und Nutzung von Geräten und Diensten läuft in UPnP und DPWS in gleichen Phasen ab. Dieser Ablauf ist unabhängig vom Framework und soll daher ebenfalls für die Erläuterung der WODA-Konzepte verwendet werden. Die Phasen lauten Addressing, Discovery, Description, Control, Eventing und Presentation (Abbildung 6.3). Die Addressing/Naming-Phase stellt sicher, dass ein Gerät eine eigene eindeutige IP-Adresse und einen eindeutigen Namen besitzt, Discovery regelt das Bekanntgeben und Auffinden von Diensten im Netzwerk, in der Description-Phase können Geräteparameter und Dienstbeschreibungen ausgetauscht wer-

den. Die Control- und Eventing-Phase regeln die Steuerung der Dienste sowie das Abonnieren und Versenden von Ereignissen, Presentation erlaubt die Benutzung eines Dienstes über einen Webbrowser. Die Phasen sind damit bis auf Addressing/Naming identisch zu UPnP und zu DPWS.

Die Abbildung 6.3 zeigt für UPnP, DPWS und WODA die Protokolle, die in den einzelnen Phasen zum Einsatz kommen. Den WODA-Stack zeigt Abbildung 6.4. Die Phasen und Protokolle werden in den folgenden Abschnitten für WODA erläutert.

6.2 Adressierung (Addressing/Naming)

Die Addressing-, Naming- und Discovery-Phase werden in WODA mit den Zeroconf-Protokollen realisiert (Abs. 2.5). Zeroconf baut auf das Domain Name System (DNS) auf und stellt damit eine etablierte und internetweit funktionierende Lösung für diese Problematik dar.

Die Addressing-Phase stellt sicher, dass ein Gerät eine eindeutige IP-Adresse erhält. Sie funktioniert analog zu UPnP: Steht kein *Dynamic Host Configuration Protocol* (DHCP) Server [55] zur Verfügung, konfiguriert Zeroconf eine zufällige IP-Adresse aus dem *link-local* Bereich und überprüft mittels *Address Resolution Protocol* (ARP), ob die Adresse im lokalen Netzwerk eindeutig ist [56].

Im Gegensatz zu UPnP und DPWS geht Zeroconf in dieser ersten Phase noch einen Schritt weiter und stellt ebenfalls sicher, dass ein Gerät einen eindeutigen Namen erhält. Dazu setzt das Gerät eine Multicast-DNS-Nachricht ab, die den gewählten Namen an die IP-Adresse bindet. Alle DNS-Einträge sind nach dem gleichen Muster aufgebaut (Name, Record-Typ und Wert). Angenommen ein Gerät hat sich mittels AutoIP die IP-Adresse *192.168.2.44* zugewiesen und bindet nun den Namen *windmill.local.* an diese Adresse. Der dazugehörige DNS-Eintrag sieht dann wie folgt aus:

```
windmill.local.    A    192.168.2.44
```

Der Record-Typ *A* verweist darauf, dass es sich bei dem Namen um eine Domain und bei dem Wert um eine IP-Adresse handelt. Je nach Infrastruktur befindet sich dieser DNS-Eintrag auf einem zentralen DNS-Server oder, bei Abwesenheit eines solchen, lokal bei allen anderen Diensten und Dienstnutzern.

Die Verwendung eines eindeutigen Namens bietet zwei Vorteile: Für Menschen ist er leichter zu merken als eine IP-Adresse, und Anwendungen, die HTTP-Anfragen an einen namensbasierten Host senden, funktionieren auch dann weiterhin, wenn sich die IP-Adresse ändern sollte. Beides

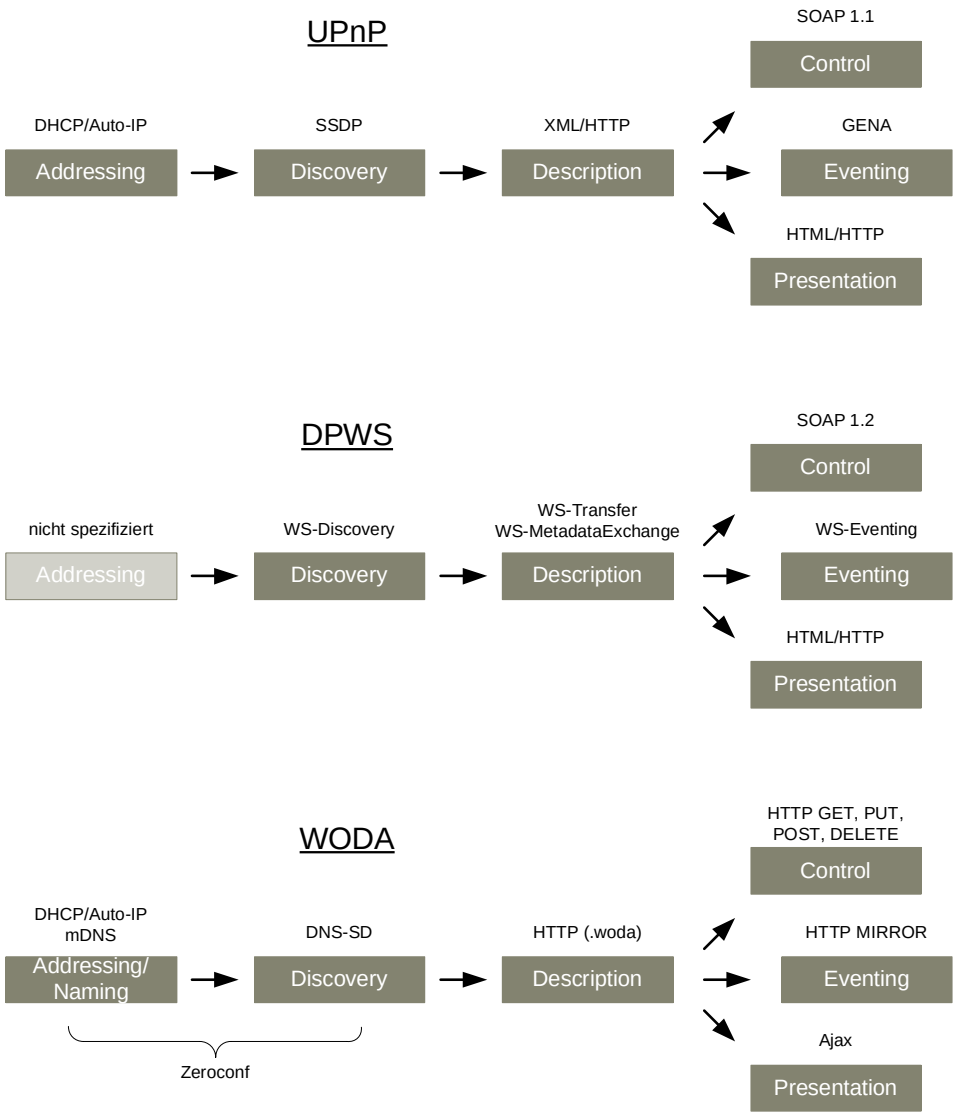

Abbildung 6.3: Phasen für die Integration und Nutzung von UPnP-, DPWS- und WODA-Diensten bzw. -Geräten

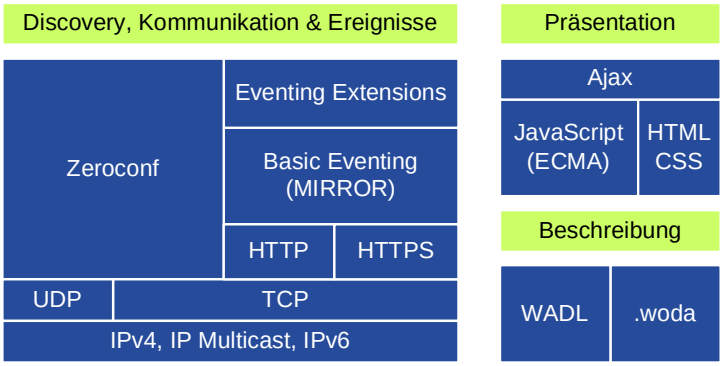

Abbildung 6.4: Der WODA-Stack

ist sowohl in UPnP als auch in DPWS nicht gegeben.

6.3 Ankündigung und Entdeckung (Discovery)

Die Discovery-Phase regelt sowohl die Bekanntgabe von Diensten als auch die Suche nach ihnen. Zeroconf verwendet mit DNS Service Discovery (DNS-SD) die Infrastruktur von DNS sowie existierende DNS Resource Records, um Dienste bekanntzugeben und zu finden. Für die Suche werden PTR- und SRV-Anfragen an einen DNS-Server gesendet. Steht dieser nicht zur Verfügung, kann, wie bereits im Abschnitt 2.5 erwähnt, über Multicast mit mDNS das lokale Netzwerk abgefragt werden.

Diensttypen und Dienstinstanzen sind DNS-Einträge, die einen bestimmten Aufbau besitzen. Diensttypen sind, wie auch Domainnamen, hierarchisch aufgebaut. Sie sind jeweils durch einen Punkt voneinander getrennt und sind von rechts nach links zu lesen. Es hat sich bewährt, Diensttypen mit einem Unterstrich voran zu notieren, um sie von Domainnamen visuell abzugrenzen. Auf die Auswertung einer DNS-Anfrage hat dieses jedoch keine Auswirkung, da bei DNS ausschließlich Namen und Typen auf Werte abgebildet werden. Ein FTP-Diensttyp wird beispielsweise als *_ftp._tcp* notiert, ein Diensttyp, der das *Internet Printing Protocol* (IPP) unterstützt, als *_ipp._tcp* usw. Beide stellen damit Subdiensttypen von TCP dar.

Um einen Diensttyp innerhalb einer Domain zu spezifizieren, muss der Name als *Service-Type.Domain* angegeben werden. So steht beispielsweise der Name *_ipp._tcp.md.local.* für die Drucker, die sich innerhalb der Domain *md.local.* befinden und IPP unterstützen.

Um schließlich einen konkreten Dienst zu identifizieren, werden Dienstinstanznamen benötigt, die ebenfalls links neben dem Diensttyp und der Domain notiert werden. Dienstinstanznamen besitzen damit folgenden Aufbau:

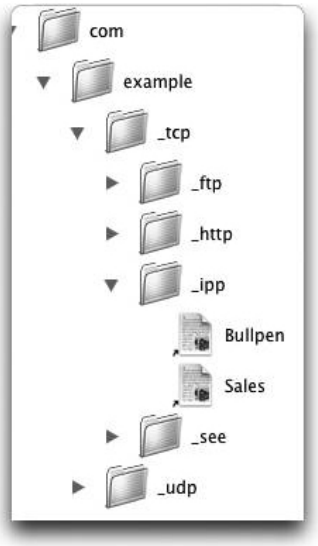

Abbildung 6.5: Auflisten von typisierten Dienstinstanzen innerhalb bestimmter Domains mit Zeroconf [54]

InstanceName.ServiceType.Domain

Instanznamen (z. B. *Drucker im Raum 1206*) unterliegen keinen Beschränkungen hinsichtlich ihres Zeichensatzes. Sie sind explizit als Repräsentation für einen Benutzer gedacht, der Dienste eines bestimmten Typs auflisten lassen möchte. Das Auflisten von typisierten Dienstinstanzen innerhalb einer bestimmten Domain war ursprünglich der Grund, die Namen analog zu Domainnamen hierarchisch aufzubauen. Der durch Zeroconf realisierte Ansatz ist damit gleichzeit für Benutzer (Abbildung 6.5) als auch für Programme benutzbar. Existierende DNS-Implementierungen können weiterhin verwendet werden, sie sind von den genannten Vereinbarungen nicht betroffen, da sie lediglich mit Namen, Record-Typen und Werten, also generisch arbeiten.

Da HTTP innerhalb von WODA das zentrale Anwendungsprotokoll ist, werden alle WODA-Diensttypen unterhalb des HTTP-Diensttyps notiert: *_woda._http._tcp*. Anhand des folgenden Beispiels soll die Suche nach Diensten demonstriert werden. Dazu wird eine DNS-Datenbank mit folgenden existierenden Einträgen vorausgesetzt:

_webcam._woda._http._tcp.local.	PTR	Raum\ 121._webcam. _woda._http._tcp.local.
_webcam._woda._http._tcp.local.	PTR	Raum\ 133._webcam. _woda._http._tcp.local.
Raum\ 121._webcam._woda._http._tcp.local.	SRV	md.local., 4900
Raum\ 133._webcam._woda._http._tcp.local.	SRV	windmill.local., 5000
md.local.	A	192.168.2.10
windmill.local.	A	192.168.2.44

Weiterhin wird hier angenommen, dass ein Webcam-Diensttyp für WODA existiert und sich zwei dieser Webcams in der lokalen Domain *.local.* befinden.

Ein Dienstnutzer, der nach Diensten suchen möchte, die den Webcam-Diensttyp implementieren, sendet eine PTR-Anfrage mit dem Namen *_webcam._woda._http._tcp.local.* und erhält als Ergebnis eine Liste mit Dienstinstanzen. Im Beispiel sind das die Einträge *Raum\ 121._webcam._woda._http._tcp.local.* und *Raum\ 133._webcam._woda._http._tcp.local.*.

PTR-Anfragen (PTR=Pointer) werden im Internet normalerweise für Reverse-Lookups verwendet, wo sie zu einer gegebenen IP-Adresse die zugehörigen Namen ermitteln. PTR sind die einzigen DNS-Abfragen, die nicht *ein* Ergebnis, sondern eine Liste von Ergebnissen zurückliefern. Deshalb sind sie für die Suche nach einer Liste von Dienstinstanzen geeignet, die einem Diensttyp zugeordnet sind. Aus DNS-Sicht ist es auch hier wiederum irrelevant welche Semantik tatsächlich hinter den Namen und Werten stecken.

In einem zweiten Schritt, oder falls die Dienstinstanz von vornherein bekannt war, kann mit einer SRV-Anfrage ein Dienstinstanzname in einen Host und Port aufgelöst werden. Für den Dienstinstanznamen *Raum\ 121._webcam._woda._http._tcp.local.* lautet der Host beispielsweise *md.local.* und der Port *4900*. Damit kennt der Dienstnutzer die erste Ressource (Wurzelressource) des Dienstes. Mit einer HTTP-GET-Anfrage kann er eine Repräsentation anfordern. Für das Beispiel lautet die URL dieser Ressource *http://md.local:4900/*. Das Absetzen dieser Anfrage hat eine weitere DNS-Anfrage zur Folge, die die zum Host zugehörige IP-Adresse, hier *192.168.2.10*, ermittelt.

Die Bekanntgabe eines Dienstes im Netzwerk wird realisiert, indem mittels DNS die o. g., bereits erläuterten DNS-Einträge erzeugt werden. Zeroconfs stärkster Vorteil ist die Ausnutzung des bereits existierenden und etablierten Domain Name Systems. Es sind mehrere (auch freie) Implementierungen verfügbar (z. B. *Bonjour* für Java[1], *Avahi* für Linuxsysteme[2] usw.)

[1] `http://www.apple.com/support/downloads/bonjourforwindows.html`
[2] `http://avahi.org/`

6.4 Beschreibung (Description)

Dienstbeschreibungen sind, wie im Abschnitt 4.2.2 erläutert, ein wichtiger Bestandteil serviceorientierter Frameworks. Sie können zur Entwicklungszeit eingesetzt werden, um Code für Dienst und Dienstnutzer zu erzeugen. Anwendungen, die in Abhängigkeit von Beschreibungsparametern arbeiten, können sie zur Laufzeit dynamisch inspizieren.

Neben der reinen Dienstbeschreibung können für Dienste, die Geräte repräsentieren, zusätzlich Geräteparameter als Teil der Beschreibung angegeben werden. WODA adressiert beide Kategorien, die der Dienst- und die der Gerätebeschreibung. Dienstbeschreibungen werden mit der *Web Application Description Language* (WADL) (Abs. 2.6) realisiert, und für die Gerätebeschreibung definiert WODA ein eigenes XML-Schema.

6.4.1 Dienstbeschreibung

Auf den ersten Blick ist eine Beschreibungssprache für HTTP-basierte Dienste gar nicht notwendig, da REST die *gleiche* Schnittstelle für *alle* Komponenten fordert und daher die Schnittstellenbeschreibung für jede Ressource gleich wäre.

Für die Generierung von Code bzw. für die formelle Definition eines bestimmten Ressourcentyps kann eine Beschreibungssprache jedoch sinnvoll sein, da beispielsweise nicht alle Ressourcen auch alle HTTP-Methoden unterstützen müssen. Hier ließen sich vordefinierte Fehlermeldungen (HTTP-Statuscodes) festlegen, die von einem Codegenerator erzeugt werden könnten. Bei der Verwendung von XML-Schema als Datenbeschreibungssprache könnte automatisch ein Validierer eingesetzt werden usw.

Es existiert bisher kein Standard, um REST-basierte Dienste formal zu beschreiben. Dennoch gibt es einige Ansätze, die innerhalb des W3Cs diskutiert werden[3]. Von diesen stellt WADL einen geeigneten Ansatz für WODA dar, da es viele Möglichkeiten zur Beschreibung von Ressourcen anbietet, es in XML notiert ist, da bereits ein Codegenerator (für Clients) existiert und es erweiterbar hinsichtlich der HTTP-Methoden ist. Letzteres ist eine wichtige Anforderung, um Ressourcen beschreiben zu können, die den im Abschnitt 6.6 vorgestellten Ereignismechanismus verwenden, da dieser eine neue HTTP-Methode definiert.

Listing 6.1 zeigt einen Ausschnitt aus einem WADL-Dokument. Die Beschreibung bezieht sich auf eine Gruppe von Ressourcen, die dem URL-Pfad *range/{id}* genügen (Zeile 1). Der Name einer hiermit adressierten Ressource könnte beispielsweise *range/7uft2s* lauten, da der URL-Parameter *id* als String spezifiziert ist (Zeile 2). Eine GET-Anfrage an diese Ressource liefert ein XML-Dokument mit einem *ub:Range* als Wurzelelement zurück (Zeilen 3–7). Ein PUT-

[3] `http://lists.w3.org/Archives/Public/public-web-http-desc/`

Listing 6.1: Ausschnitt aus einem WADL-Dokument

```
1  <resource path="range/{id}">
2      <param name="id" style="template" type="xs:string"/>
3      <method name="GET">
4          <response>
5              <representation mediaType="text/xml" element="ub:Range"/>
6          </response>
7      </method>
8      <method name="PUT">
9          <request>
10             <representation mediaType="text/xml" element="ub:Range"/>
11         </request>
12     </method>
13     ...
14 </resource>
```

Aufruf kann dagegen in der Anfrage mit einem solchen Element versehen werden (Zeilen 8–12). Das zugehörige XML-Schema ist ebenfalls Teil des WADL-Dokumentes.

6.4.2 Gerätebeschreibung

WODA definiert ein eigenes XML-Schema für die Gerätebeschreibung, siehe Anhang C. Eine solche Gerätebeschreibung referenziert bzw. verlinkt die zum Gerät gehörigen Dienste und spezifiziert Geräteparameter wie Hersteller, Modellnummer, weiterführende Links usw. Für die Angabe der Geräteparameter werden die beiden Elemente *ThisDevice* und *ThisModel* in Anlehnung an DPWS wiederverwendet (Abs. 2.8).

Während die Dienste eines Gerätes unterschiedliche Beschreibungen besitzen, ist umgekehrt die Gerätebeschreibung für jeden Dienst auf diesem Gerät die gleiche. In WODA werden während der Discovery-Phase, wie oben erläutert, Wurzelressourcen von Diensten, nicht jedoch von Geräten, ermittelt. Es muss demnach ein Weg gefunden werden, um von der Wurzelressource eines Dienstes zur Beschreibung des Gerätes zu gelangen, zu dem dieser Dienst gehört. Dazu definiert WODA eine spezielle Ressource, die jedes Gerät anbieten muss und die die Gerätebeschreibung enthält. Diese Ressource besitzt mehrere Ressourcenbezeichner – für jeden Dienst einen in der Form *http://{host}:{port}/.woda*.

Abbildung 6.6 zeigt ein WODA-Gerät mit den Wurzelressourcen dreier Dienste, so wie sie mittels Discovery gefunden werden können. Die vierte Ressource repräsentiert die Gerätebeschreibung, die für alle drei Dienste die gleiche ist und die demnach durch drei unterschiedliche Ressourcenbezeichner identifiziert werden kann.

Nach einer Dienstsuche (PTR- und SRV-Anfrage) sind der Host und Port eines Dienstes bekannt. Aus diesen Werten lässt sich die URL der Ressource für die zugehörige Gerätebeschrei-

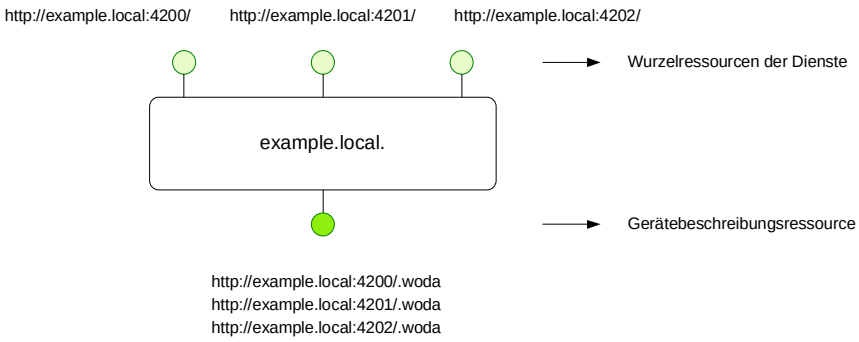

Abbildung 6.6: Wurzelressourcen dreier Dienste und gemeinsame Ressource für die Gerätebeschreibung

bung konstruieren und mit einer HTTP-GET-Anfrage abrufen. Der Vorteil dieses Ansatzes ist, dass kein zusätzliches Protokoll benötigt wird wie es beispielsweise bei DPWS mit dem WS-Transfer- und dem WS-MetadataExchange-Protokoll der Fall ist.

6.5 Steuerung (Control)

In der Steuerungsphase werden Dienste durch Dienstnutzer verwendet. In WODA geschieht dieses durch das Senden von HTTP-Nachrichten an Dienstressourcen. Um eine Ressource zu lesen, also eine Repräsentation zu erhalten, initiiert der Client eine *GET*-Anfrage an den Server. Wie bei allen HTTP-Anfragen enthält diese die URL der Ressource (aufgesplittet in Host und Pfad). Optional kann mit dem *Accept*-Header das gewünschte Datenformat der Repräsentation angegeben werden. Um den Inhalt einer Ressource und damit den Zustand der WODA-Komponente zu ändern, sendet ein Client eine *PUT*-Anfrage zusammen mit der neuen Repräsentation an den Server. Mit *DELETE* lässt sich schließlich eine Ressource löschen.

Um eine neue Ressource explizit anzulegen, wird mittels *PUT* eine Repräsentation an eine noch nicht vorhandene Ressource gesendet. „Nicht vorhanden“ bedeutet, dass ein *GET*-Aufruf an die gleiche URL zuvor mit einem *404-Not-Found*-Fehlercode beantwortet worden wäre. Der Server kann die Anfrage ablehnen oder aber die Ressource mit der gewünschten URL und Repräsentation erzeugen (Statuscode *201 CREATED*).

Die *POST*-Methode nimmt eine Sonderstellung unter den HTTP-Methoden ein, da ihr so gut wie keine Semantik zugewiesen ist. HTTP spezifiziert lediglich, dass die Ressource, die die Anfrage identifiziert, die Daten verarbeiten können muss. Die Ressource kann daher als *Prozessor* verstanden werden, welcher die Daten interpretiert. Das Ergebnis ist abhängig von der Arbeits-

weise des Prozessors. Es können beispielsweise neue Ressourcen erzeugt, vorhandene geändert oder einfach nur neue Daten berechnet und zurückgesendet werden, ohne den Zustand der Komponente zu ändern. Wie genau eine solche Prozessorressource arbeitet und was für Daten sie benötigt, fällt in den Definitionsbereich des Dienstes oder eines aufgesetzten Anwendungsprotokolls. Aufgrund dieser sehr weit gefassten Semantik können mit POST beliebige Methoden realisiert werden. Insbesondere wird es häufig zum Tunneln anderer Protokolle wie etwa SOAP verwendet.

WODA beschränkt daher die Nutzung von POST auf ein Minimum. Die Herausforderung beim Entwurf eines WODA-Dienstes besteht also darin, möglichst viele *CRUD*[4]-Ressourcen und wenig Prozessor-Ressourcen zu modellieren. Jede Prozessor-Ressource erzeugt zusätzlichen Aufwand, da deren Semantik erst erklärt werden muss.

6.6 Ereignisse (Eventing)

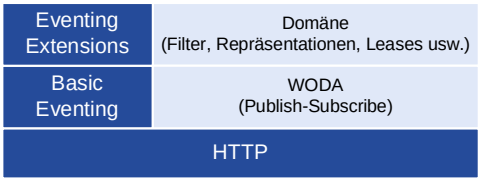

Abbildung 6.7: Eventing-Stack von WODA

Neben der Steuerung von Geräten bzw. Diensten sind Dienstnutzer häufig an in Diensten auftretenden Ereignissen interessiert. UPnP und DPWS definieren jeweils einen Publish-Subscribe-Mechanismus, mittels welchen Statusvariablen (UPnP) oder Operationen (DPWS) abonniert werden können. WODA definiert ebenfalls einen Publish-Subscribe-Mechanismus. Mit diesem können Ressourcen abonniert werden. Der Eventing-Stack teilt sich auf in zwei Ebenen (Abbildung 6.7). Auf unterer Ebene definiert WODA den generischen Publish-Subscribe-Mechanismus (Basic Eventing), die obere Ebene ist domänenspezifischen Anwendungsprotokollen vorbehalten, die zusätzliche Erweiterungen wie Filter, Repräsentationsformate oder Lease-Mechanismen definieren wollen (Eventing Extensions).

Ereignisse treten, wie im Abschnitt 4.2.3 erläutert, unabhängig davon auf, ob jemand sie beobachtet oder ob sie zuvor über einen Publish-Subscribe-Mechanismus abonniert wurden.

[4]CRUD stammt aus dem Bereich der Datenbanken, wo es für die grundlegenden Datenbankoperationen *create*, *read*, *update* und *delete* steht

Um als Dienstnutzer jedoch über Ereignisse informiert zu werden bzw. an die damit generierten oder geänderten Daten zu gelangen, werden geeignete Mechanismen benötigt.

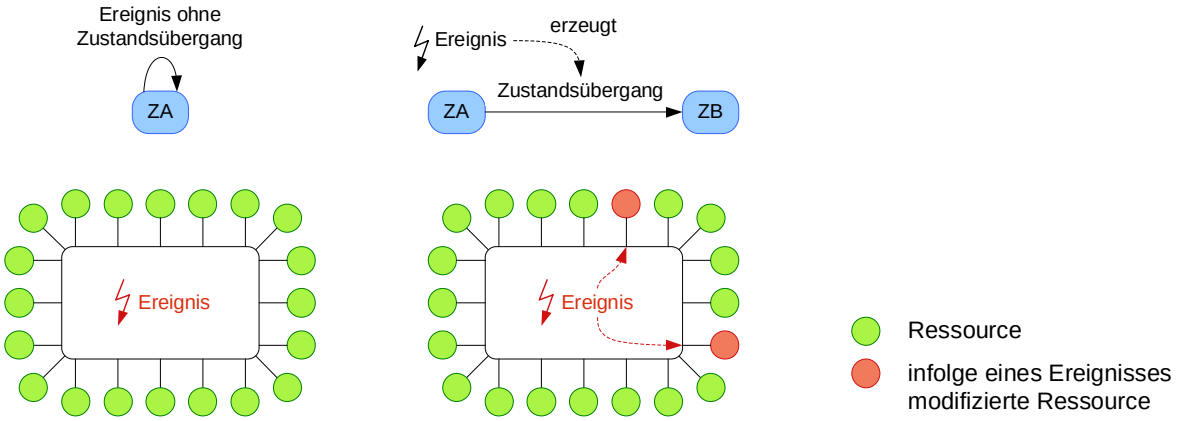

Abbildung 6.8: Änderungen von Ressourcen infolge eines Ereignisses

Ein *endlicher Zustandsautomat* ist ein geeignetes Modell, um die Zusammenhänge zwischen Ereignissen, Zuständen, Ein- und Ausgaben in einem System abzubilden. Demnach befindet sich ein System zu einem bestimmten Zeitpunkt immer in einem definierten Zustand. Dieser ist gekennzeichnet durch die Belegung der freien Variablen mit konkreten Werten. Durch Ereignisse (bzw. Eingaben) werden Zustandsübergänge ausgelöst, die entweder im gleichen oder in einem neuen Zustand münden.

In WODA werden nun, wie weiter oben bereits erläutert, alle Daten als Ressourcen modelliert, die zusammen den Zustand der entsprechenden Komponente repräsentieren. Ein definierter Zustand ist demnach gekennzeichnet durch die Werte aller Ressourcen zu einem bestimmten Zeitpunkt. Diese Modellierung impliziert, dass sich bei einer Zustandsänderung auch mindestens eine Ressource ändern muss (Abbildung 6.8). Um Ereignisse bekanntzugeben oder überwachen zu können, müssen demnach die entsprechenden Ressourcen, die von den Ereignissen betroffen sind, bekanntgegeben oder überwacht werden. Es handelt sich hierbei um indirekte Ereignisbeobachtungen, da nur über die Beobachtung von Ressourcen auf Ereignisse geschlossen werden kann. Ereignisse, die keine Zustandsänderung zur Folge haben, können demnach auch nicht beobachtet werden. Es liegt also beim Dienstentwickler, den Dienst so zu entwerfen, dass Ereignisse beobachtbar sind. Eine generische Möglichkeit ist, das Ereignis selbst als Ressource zu modellieren. Die entsprechende Repräsentation könnte den Zeitpunkt des Ereignisses oder einen fortlaufenden Zähler enthalten. Tritt das Ereignis auf, ändert sich auch die Repräsentation.

Die Beobachtung von Ereignissen bzw. Ressourcen mittels des *Pull*-Patterns ist mit der GET-

Methode in HTTP bereits eingebaut. Hierzu erübrigen sich weitere Betrachtungen. WODA definiert einen zusätzlichen *Push*-basierten Mechanismus. Dazu wendet es das Publish-Subscribe-Pattern an, indem es die neue HTTP-Methode *MIRROR* und den neuen HTTP-Header *Mirror-Sink* definiert. Mit MIRROR kann ein Dienstnutzer eine Ressource bei einem Server abonnieren. Der Dienstnutzer erhält in der Folge immer dann eine Kopie der entfernten Ressource, wenn sich diese ändert.

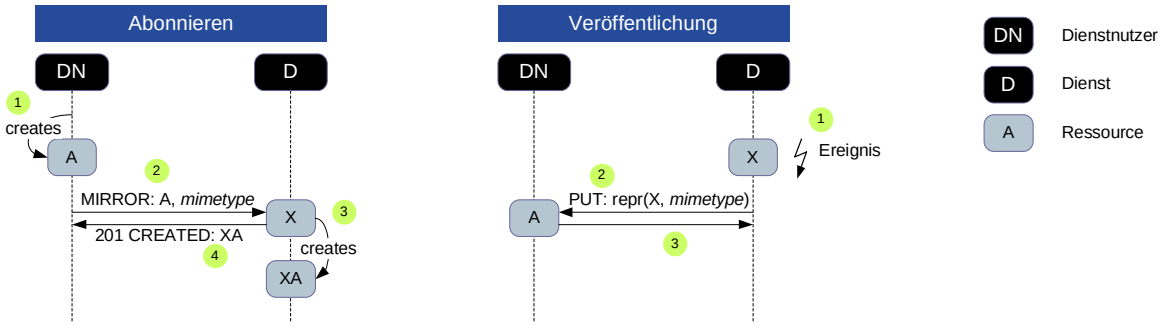

Abbildung 6.9: Abonnieren und Veröffentlichen von Ressourcen

In Abbildung 6.9 ist der Dienstnutzer (Komponente *DN*) an der Ressource *X* interessiert, die durch das Ereignis modifiziert wird oder aber das Ereignis selbst repräsentiert. Inhaber der Ressource ist ein Dienst (Komponente *D*). Um die Ressource *X* abonnieren zu können, muss der Dienstnutzer zunächst eine eigene Ressource *A* auf seinem System erzeugen und reservieren, welche eine Kopie der entfernten Ressource *X* repräsentiert. Im nächsten Schritt sendet der Dienstnutzer eine MIRROR-Anfrage an *X*. Diese enthält im MirrorSink-Header die URL der zuvor erzeugten Ressource *A*. Zur weiteren Spezifizierung der Anfrage können die Header herangezogen werden, die auch bei einer GET-Anfrage überlichweise verwendet werden: *Accept*, *Accept-Charset*, *Accept-Encoding* sowie *Accept-Language*. Beispielsweise lässt sich dadurch ein für die Repräsentation gewünschtes Datenformat einstellen.

Akzeptiert der Dienst das Abonnement, erzeugt er eine neue Ressource *XA*, die das Abonnement repräsentiert, und antwortet mit einem HTTP *201 CREATED* Statuscode. Dieser verweist auf das erfolgreiche Erzeugen einer neuen Ressource und gibt die URL zu *XA* im *Location*-Header zurück. Folgende, in HTTP bereits existierende Statuscodes sollten zurückgesendet werden, wenn der Dienst das Abonnement nicht akzeptiert:

405	Method Not Allowed	Die Ressource erlaubt die *MIRROR*-Methode nicht.
503	Service Unavailable	Dienst nicht verfügbar. Kann verwendet werden, falls bereits zu viele Dienstnutzer die Ressource abonniert haben.

Ändert sich infolge eines Ereignisses der Zustand der Ressource X, schickt der Dienst eine PUT-Nachricht an die URL, die der Abonnent im MirrorSink-Header angegeben hat. Diese enthält die neue Repräsentation, wobei der Dienst die Anfrageparameter (*Accept*-Header) der vorausgegangenen MIRROR-Anfrage beachten muss. Der Body der Nachricht besitzt demnach den gleichen Inhalt, als wenn es sich dabei um eine Antwort auf eine GET-Anfrage zum selben Zeitpunkt handeln würde. Das Veröffentlichen der Ressource wird für jeden Abonnenten wiederholt.

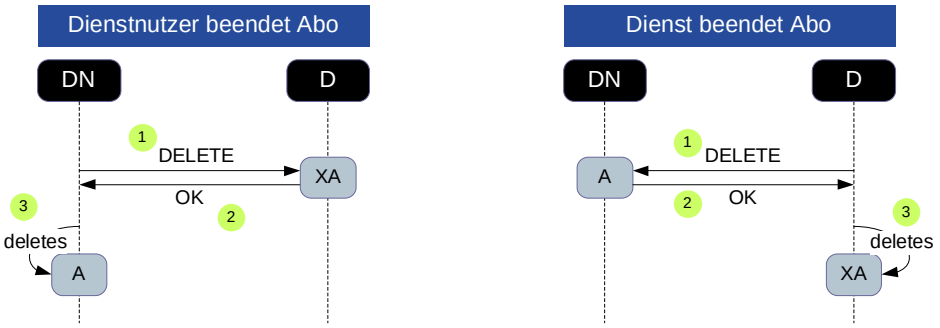

Abbildung 6.10: Beenden von Abonnements

Abonnements haben auf WODA-Ebene eine unbestimmte Gültigkeitsdauer. Sie können jedoch von beiden Seiten beendet werden. Die Vorgehensweise ist sehr intuitiv. Möchte der Dienstnutzer keine weiteren Nachrichten erhalten, sendet er eine DELETE-Nachricht an die für sein Abonnement erstellte Ressource XA (Abbildung 6.10). Der Dienst muss daraufhin das Abonnement entfernen und darf keine weiteren Nachrichten an den Dienstnutzer schicken. Möchte der Dienst dagegen selbst das Abonnement beenden, beispielsweise weil er seine Arbeit beendet oder weil die gespiegelte Ressource X nicht mehr existiert, schickt er eine DELETE-Nachricht an die Senkenressource A, die zuvor vom Dienstnutzer erstellt wurde.

Der in WODA verwendete Publish-Subscribe-Mechanismus zeichnet sich durch seine Einfachheit aus, der mit nur einer weiteren HTTP-Methode und einem weiteren HTTP-Header auskommt. Alle anderen Aufgaben lassen sich mit den gängigen Mitteln des HTTP-Protokolls

lösen. Weiterhin ist dieser Ansatz vollständig REST-konform, da alle Nachrichten zustandslos sind und die Datenelemente, auf denen die Methoden arbeiten, alle als eigene Ressourcen modelliert wurden (Abonnements, Quell- und Zielressourcen).

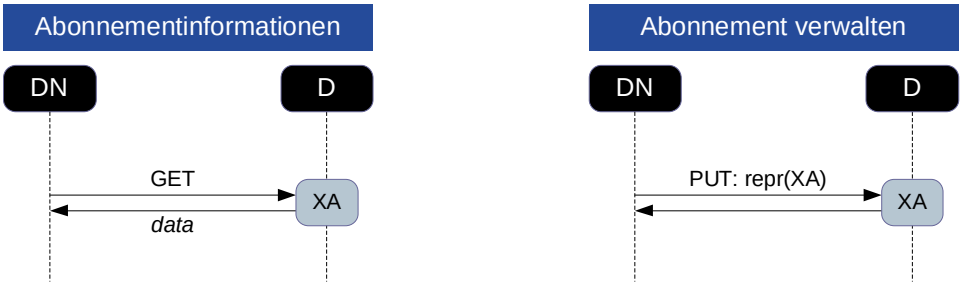

Abbildung 6.11: Beispiel für eine Eventing-Extension: Verwaltung von Abonnements

WODAs Basic-Eventing-Ebene stellt eine domänenübergreifende Möglichkeit dar, ereignisbasierte Nachrichten nach dem Publish-Subscribe-Pattern zu realisieren. Daneben steht es Anwendungen und Domänen auf der Eventing-Extension-Ebene frei, Erweiterungen zu definieren, beispielsweise:

- **Änderung einer Ressource** WODA definiert nicht, was Änderung einer Ressource bedeutet. Vielmehr ist dieses WODA auch gar nicht möglich, da die Definition hierüber immer durch die Modellierung des Dienstes erfolgt. Ein Temperaturdienst, der beispielsweise die aktuelle Raumtemperatur über eine Ressource zur Verfügung stellt, wird die Temperatur mit einem Sensor messen, der eine bestimmte Auflösung im Messbereich aufweist (z. B. 1/1000). Dennoch ist es die Implementierung, welche darüber entscheidet, ab wann sich die Ressource ändert (z. B. nur im Ganzzahlbereich).
- **Gültigkeitsdauer** Abonnements haben in WODA eine unbestimmte Gültigkeitsdauer. Erweiterungen können aus der unbestimmten eine bestimmte oder eine Mindestgültigkeitsdauer machen, indem sie zusätzliche Header oder Nachrichtenformate definieren und beispielsweise das Lease-Konzept anwenden. Oder die Gültigkeit des Abonnements wird auf eine einmalige Ereignisbenachrichtigung beschränkt. Unabhängig von etwaigen Erweiterungen wird das Abonnement jedoch immer erst mit dem Entfernen der Ressource, die das Abonnement repräsentiert, beendet bzw. ungültig.
- **Filter** Erweiterungen können zusätzlich Filtermechanismen spezifizieren, die die zu übertragenen Datenmengen reduzieren oder nur dann eine Ereignisnachricht ausliefern, wenn sie bestimmte Kriterien erfüllt. Hier können Ansätze wie die *Rule Markup Language* (Rule-

ML) oder XPath (siehe auch *Filter* in WS-Eventing, Abschnitt 2.3.7) zum Einsatz kommen.

- **Abonnement-Repräsentation** Erweiterungen können festlegen, wie Abonnement-Ressourcen repräsentiert werden. Diese könnten beispielsweise den Zeitpunkt der Erstellung, Gültigkeitsdauer und Filtereinstellungen enthalten. Da ein Abonnement durch eine Ressource modelliert wird, kann diese dann mittels einer PUT-Anfrage durch den Abonnenten geändert werden (Abbildung 6.11). Dadurch lässt sich beispielsweise leicht das Lease-Konzept realisieren. Voraussetzung für alle genannten Beispiele ist, dass sowohl Dienst als auch Dienstnutzer die jeweilige Erweiterung verstehen.

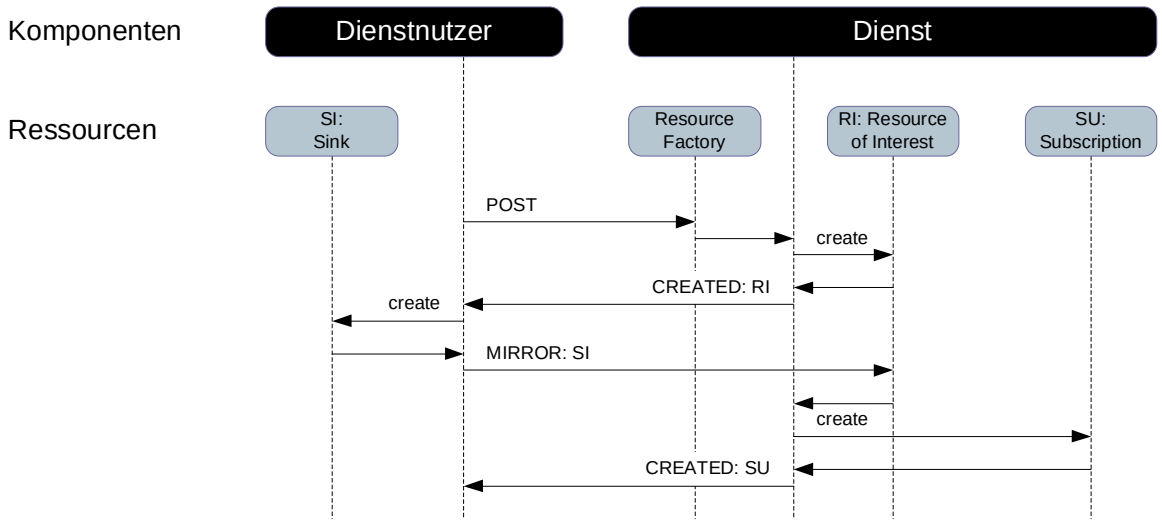

Abbildung 6.12: Verwendung einer *Resource Factory* für benutzerdefinierte, abonnierbare Ressourcen

Da der Ressourcenentwurf in der Hand des Entwicklers liegt, können in der Folge auch nur die Ressourcen abonniert werden, die der Entwickler als abonnierbar deklariert hat. Das führt in Situationen zu Problemen, in denen benutzerspezifische abonnierbare Ressourcen auf Dienstnutzerseite erwünscht sind. Eine elegante Lösung bietet die Verwendung einer speziellen Prozessor-Ressource, an die ein Dienstnutzer Daten schicken kann, um eine für ihn spezifische Ressource erzeugen zu lassen, die er anschließend abonnieren kann. Ressourcen, die dieses Pattern unterstützen, sollen im Folgenden *Resource Factory* genannt werden (Abbildung 6.12).

Da der Ereignismechanismus von WODA vollständig REST-konform ist, profitiert er von den Eigenschaften, die Ressourcen, URIs und HTTP mit sich bringen. So lassen sich beispielsweise Abonnements auf einer anderen Komponente verwalten als auf derjenigen, die die abonnierten

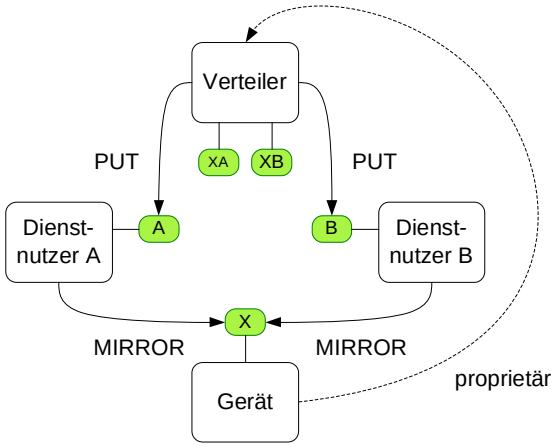

Abbildung 6.13: Auslagerung der Abonnementverwaltung und Nachrichtenveröffentlichung auf eine externe Verteilerkomponente

Ressourcen besitzt. Für einen Dienstnutzer ist diese Art Outsourcing transparent und vor allem für ressourcenbeschränkte Geräte geeignet, die sich nicht um Abonnementverwaltung und Ereignisverteilung kümmern wollen oder können. Die Kommunikation zwischen Dienst und Verteiler kann über einen proprietären Mechanismus erfolgen (Abbildung 6.13). WODA unterstützt damit die beiden im Abschnitt 4.2.3 geforderten Ereignismodelle *Peer-to-Peer* und *Vermittler*.

6.7 Präsentation (Presentation)

Die Vielzahl und Vielfältigkeit von Geräten erfordert gerade in Mensch-zu-Maschine-Szenarien eine einfache Möglichkeit zur Konfiguration und Steuerung. Der Webbrowser stellt den Quasi-Standard für einen generischen, grafischen Client dar, mit dem diese Aufgabe erfüllt werden kann. UPnP, DPWS und WODA bieten daher als Teil ihrer Gerätebeschreibungen eine optionale *PresentationURL* an. Diese kann von einem Dienstnutzer in einen Webbrowser geladen werden, und anschließend kann das Gerät über den Webbrowser gesteuert werden. Voraussetzung hierfür ist eine entsprechende Webanwendung.

Webanwendungen werden in den meisten Fällen für einen Server implementiert, auf welchem sie laufen. Hier haben sich in der Vergangenheit eine Vielzahl von Technologien wie PHP, ASP, JSP usw. durchgesetzt. Diese generieren aus Daten, die sie von einem Webbrowser in der Regel über ein ausgefülltes HTML-Formular erhalten, dynamisch HTML-Seiten, die sie wieder an den Webbrowser zurücksenden.

Die genannten Technologien sind jedoch gerade für ressourcenarme Geräte völlig ungeeignet,

da sie für den Einsatz auf leistungsfähigen Servern konzipiert sind. Zudem müssen Geräte in diesem Fall neben der frameworkspezifischen Schnittstelle noch eine zweite, zusätzliche Schnittstelle für eine Webanwendung implementieren, die mit einem Webbrowser zusammenarbeiten kann. Beide Schnittstellen implementieren jedoch den gleichen Dienst (Abbildung 6.1).

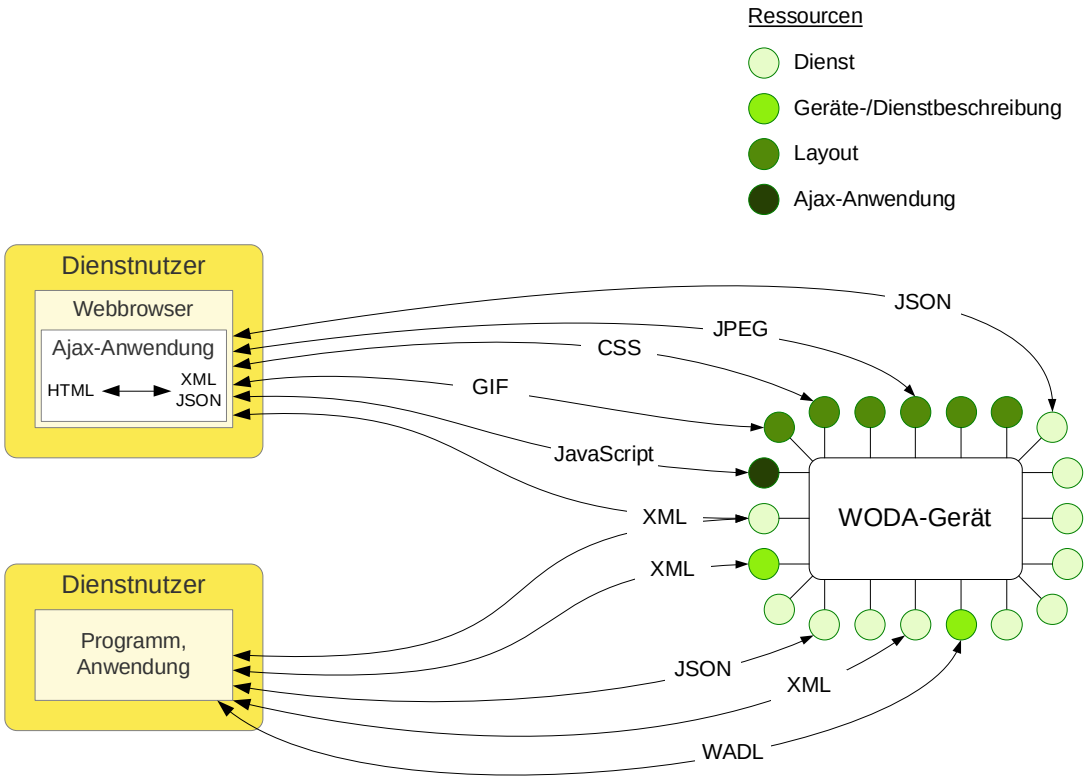

Abbildung 6.14: Verwendung der *gleichen* Schnittstelle sowohl für die Maschine-zu-Maschine- als auch für die Mensch-zu-Maschine-Kommunikation

Um diese Probleme zu lösen, favorisiert WODA den Einsatz der Ajax-Technologie. Ajax-basierte Anwendungen sind JavaScript-Anwendungen, die direkt im Webbrowser ausgeführt werden und aus reinen Daten (z. B. XML oder JSON) HTML erzeugen. Damit sind sie hervorragend für Geräte geeignet, da sämtliche Berechnungen für die Darstellung durch den Webbrowser selbst ausgeführt werden können und dadurch das Gerät entlastet wird. Außerdem lassen sich die Anwendungsressourcen der Dienstimplementierung wiederverwenden. Es ist also keine zusätzliche Schnittstelle wie bei UPnP oder DPWS notwendig (Abbildung 6.14). Sowohl der Programm-Dienstnutzer (Maschine) als auch der Webbrowser-Dienstnutzer (Mensch) können auf die glei-

chen Ressourcen zugreifen. Die dienstspezifisch zu implementierende Ajax-Anwendung ist selbst auch eine Ressource, die auf dem Gerät statisch als Datei hinterlegt werden kann und vom Webbrowser geladen wird.

Ajax erfordert die Unterstützung von JavaScript auf einem Webbrowser. Alle gängigen, modernen Webbrowser erfüllen diese Anforderung. Zudem existieren viele JavaScript-Bibliotheken (z. B. JQuery[5], MooTools[6], ExtJS[7] usw.), die von der verwendeten Browserplattform abstrahieren und somit eine browserübergreifende, einfache Entwicklung von Ajax-Anwendungen ermöglichen.

6.8 Anwendungsbeispiel

Um die vorgestellten Konzepte an einem Beispiel zu demonstrieren, wird im Folgenden ein Indoor-Lokalisierungsdienst (Ubisense-Dienst) exemplarisch umgesetzt, so wie er auch schon im Abschnitt 5.4.5 spezifiziert wurde, diesesmal jedoch nicht als WSDL-konformer Web Service, sondern als WODA-Dienst. Die WSDL des Ubisense-Dienstes bot im Wesentlichen drei Operationen an: Positionsabfrage für ein bestimmtes Tag, Positionsabfrage für einen durch zwei Punkte aufgespannten Quader (Gebiet) und die Überwachung eines Quaders.

Um einen WODA-Dienst zu entwerfen, müssen zunächst sämtliche Ressourcen identifiziert werden, die der Dienst anbieten soll. Weiterhin müssen die Methoden sowie Datenformate festgelegt werden, die in den Interaktionen ausgetauscht werden. Dieser Entwurf wird anschließend in einem WADL-Dokument spezifiziert.

Tabelle 6.1 zeigt die Ressourcen und Methoden für den Ubisense-Dienst. Das zugehörige WADL-Dokument ist im Anhang C zu finden. Das XML-Schema, welches auch schon zuvor im Abschnitt 5.4.5 zum Einsatz kam, wird hier wiederverwendet und zusätzlich um die beiden Typen *Range* und *Ranges* erweitert. Eine *Range* beschreibt die Abmessung eines Quaders (*Box*) sowie die darin enthaltenen Tags (*TagList*). *Ranges* enthält eine Liste von *Range*-Elementen (Anhang C).

Der Ubisense-Dienst spezifiziert zwei Ressourcen und zwei Ressourcengruppen: Die *tags*-Ressource repräsentiert alle Tags, die vom Dienst verwaltet werden. Mit *tag{id}* wird das Tag mit der angegebenen ID adressiert. Beide Ressourcen können lediglich mit GET gelesen werden. Eine Ressource der Gruppe *range{id}* repräsentiert jeweils ein überwachtes Gebiet. Sie kann mit MIRROR abonniert und mit DELETE gelöscht werden. Schließlich kann der Dienstnutzer mit einer PUT-Anfrage die Abmaße des Gebietes ändern. Um ein neues zu überwachendes

[5] `http://jquery.com/`
[6] `http://mootools.net/`
[7] `http://extjs.com/`

Tabelle 6.1: Ressourcen für den Ubisense-Dienst

`tags` – Repräsentiert alle Tags	
GET	Liste mit allen Tags und ihren Positionen
`tags/{id}` – Repräsentiert das Tag mit der ID *id*	
GET	liefert die Position des Tags
`ranges` – Repräsentiert alle überwachten Gebiete und fungiert gleichzeitig als *Resource Factory* (siehe S. 137), um neue Gebiete zu erzeugen	
GET	Liefert eine Liste mit allen überwachten Gebieten
POST	Anfrage, um ein neues Gebiet zu erzeugen
`range/{id}` – Repräsentiert ein Gebiet	
GET	Liefert die Geometrie des Gebietes und eine Tag-Liste mit deren Positionen
PUT	Bewirkt Änderungen an der Geometrie des Gebietes
DELETE	Löscht das Gebiet, sofern keine gültigen Abonnements für das Gebiet existieren
MIRROR	Erzeugt ein Abonnement, um Änderungen an der Tag-Liste zu verfolgen

Gebiet anzulegen, sendet der Dienstnutzer eine POST-Nachricht an die Ressource *ranges*. Das gewünschte Gebiet wird mit einem *Range*-Element definiert, wie aus dem WADL-Dokument ersichtlich ist. Akzeptiert die *ranges*-Ressource die Anfrage, erzeugt sie eine neue *range/{id}*-Ressource, beispielsweise *range/2vbd8fu*. Die *ranges*-Ressource setzt damit eine *Resource Factory* (siehe S. 137) um. Dieses ist notwendig, da dem Ubisense-Dienst die zu überwachenden Gebiete nicht von vornherein bekannt sein können, sondern erst durch Dienstnutzer erzeugt werden sollen.

Für die weitere Betrachtung wird angenommen, dass das Ubisense-Gerät neben dem Ubisense-Dienst einen weiteren Dienst (Power-Dienst) anbietet. Für diesen Fall zeigt Listing 6.2 eine mögliche Gerätebeschreibung für das Ubisense-Gerät. Neben den Geräteparametern enthält diese eine Liste mit Links auf die Wurzelressourcen der beiden Dienste und optional auf die WADL-Dokumente, die die Dienste beschreiben.

Die Abbildung 6.15 zeigt das Ubisense-Gerät mit einigen Beispielressourcen: *http://ubisense.local:5100/tag/96728* repräsentiert das Tag mit der ID *96728*, *http://ubisense.local:5100/range/509* das überwachte Gebiet mit der ID *509*, *http://ubisense.local:5100/web* ist die Wurzelressource für die Webanwendung des Ubisense-Gerätes, sie ist in der Gerätebeschreibung als *PresentationURL* ausgezeichnet, und die beiden Bezeichner *http://ubisense.local:5100/.woda* sowie *http://ubisense.local:4400/.woda* repräsentieren die Gerätebeschreibungsressource.

Listing 6.2: WODA-Gerätebeschreibung für das Ubisense-Gerät

```
<woda:Device xmlns:woda="http://www.ws4d.org/woda/dev"
    xmlns:dpws="http://schemas.xmlsoap.org/ws/2006/02/devprof">
    <woda:Services>
        <woda:Service>
            <woda:InstanceName>Labor</woda:InstanceName>
            <woda:ServiceType>_ubisense._woda._http._tcp</woda:ServiceType>
            <woda:Url>http://ubisense.local:5100/</woda:Url>
            <woda:Wadl>http://ubisense.local:5100/ubisense.wadl</woda:Wadl>
        </woda:Service>
        <woda:Service>
            <woda:InstanceName>Labor</woda:InstanceName>
            <woda:ServiceType>_power._woda._http._tcp</woda:ServiceType>
            <woda:Url>http://ubisense.local:4400/</woda:Url>
        </woda:Service>
    </woda:Services>
    <dpws:ThisModel>
        <dpws:Manufacturer>Ubisense</dpws:Manufacturer>
        <dpws:ManufacturerUrl>http://www.ubisense.de/</dpws:ManufacturerUrl>
        <dpws:ManufacturerUrl>http://www.ubisense.net/</dpws:ManufacturerUrl>
        <dpws:ModelName>Ubisense Serie 7000</dpws:ModelName>
        <dpws:PresentationUrl>http://ubisense.local:5100/web
        </dpws:PresentationUrl>
    </dpws:ThisModel>
    <dpws:ThisDevice>
        <dpws:FriendlyName>Ubisense Labor Uni Rostock</dpws:FriendlyName>
        <dpws:SerialNumber>23498-25874698-2</dpws:SerialNumber>
    </dpws:ThisDevice>
</woda:Device>
```

6.9 Vergleich

Die Web-oriented Device Architecture setzt mit Zeroconf und HTTP auf zwei internetweit etablierte Standards. Weiterhin verwendet sie mit WADL und mit dem eigenen Eventing-Ansatz zwei weitere Mechanismen, die sich sehr gut in das Umfeld der Webtechnologien und der dem WWW zugrundeliegenden REST-Architektur einfügen. Außerdem ermöglicht sie durch den Einsatz der Ajax-Technologie eine ressourcenschonende Benutzbarkeit von Geräten durch herkömmliche Webbrowser. WODA realisiert damit aus SOA-Sicht einen verbesserten Ansatz, da Dienste nur *eine* Schnittstelle unabhängig von der Art des Dienstnutzers anbieten müssen.

Im Folgenden sollen noch einige Vergleiche mit UPnP und DPWS, vor allem für die Discovery- und die Eventing-Phase, angestellt werden.

Die Abbildung von und die Beziehungen zwischen Geräten und Diensten sind in UPnP, DPWS und WODA unterschiedlich ausgeprägt (Abbildung 6.16). Während Geräte in UPnP hierarchisch aufgebaut sein können, ist die Beziehung bei DPWS flach (keine Subgeräte). WODA kennt, abgesehen von der Gerätebeschreibung, das Konzept eines Gerätes bzw. eines Gerätetyps nicht. Dieses ist auch gar nicht notwendig, da in einer SOA alle Funktionalitäten und Anwendungen als

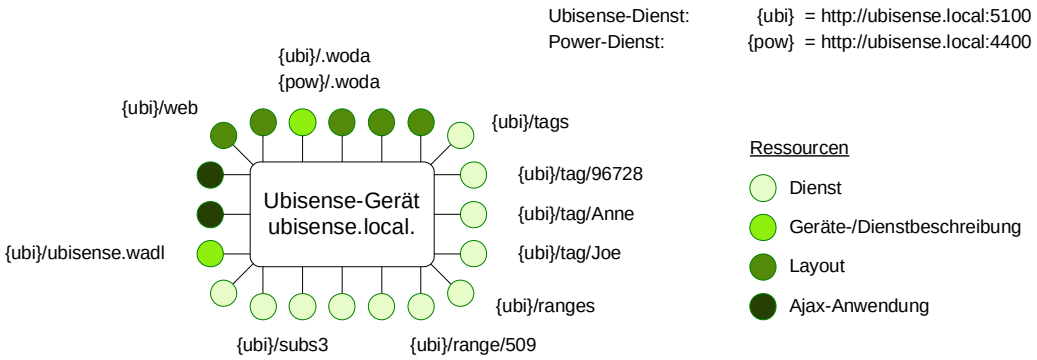

Abbildung 6.15: Ressourcen des Ubisense-Gerätes

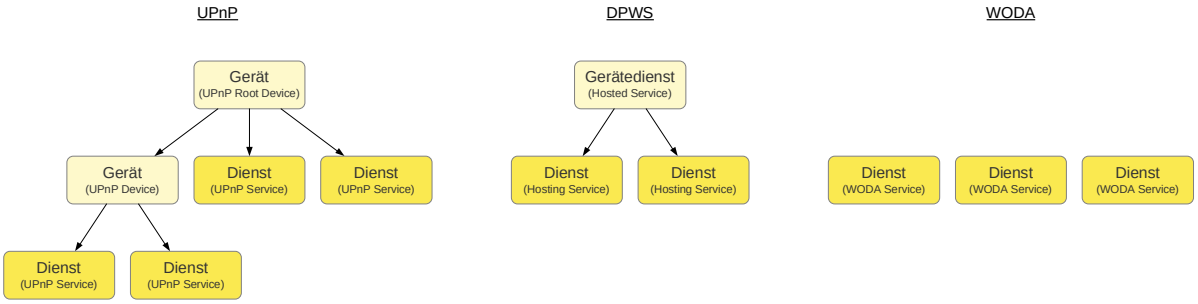

Abbildung 6.16: Abbildung von und Beziehungen zwischen Geräten und Diensten in UPnP, DPWS und WODA

Dienste abgebildet werden. Bei einer Dienstsuche wird in der Regel nach einem Dienst gesucht, der eine bestimmte Aufgabe erfüllt (Diensttyp). Der Typ gibt unter anderem Auskunft darüber, *wie* mit dem Dienst zu kommunizieren ist (Anwendungsprotokoll). Es ist dabei gleichgültig, auf welcher Hardware dieser implementiert ist. Soll dagegen ein bestimmter Dienst gefunden werden, werden Dienstinstanzen gesucht. Die Dienstinstanzen sind es letztlich, die ein bestimmtes Gerät identifizieren.

UPnP, DPWS und WODA unterstützen das im Abschnitt 4.2.1 geforderte *symmetrische Plug-and-Play* gleichermaßen. Dennoch gibt es gravierende Unterschiede (Tabelle 6.2).

Der Discovery-Vorgang ist in DPWS auf mehrere Schritte verteilt (Abs. 2.8). Bei der eigentlichen Bekanntgabe ist die Angabe von Geräte- oder Diensttypen optional. Ein Dienstnutzer muss daher immer davon ausgehen, dass die Typenliste unvollständig ist. Somit beschränkt sich das *Announcement* bei DPWS auf An- und Abwesenheitsmeldungen eines Gerätes.

WODA unterstützt neben einer typenbasierten Suche auch das Suchen nach konkreten Dienst-

Tabelle 6.2: Vergleich der Frameworks bezüglich Discovery

	UPnP	DPWS	WODA
Allgemein			
Protokoll	SSDP	WS-Discovery, WS-MetadataExchange, WS-Transfer	mDNS, DNS-SD
Bekanntgabe (*Announcement*)			
Anwesenheit	ja	ja	ja
Abwesenheit	ja	ja	ja
Lease-Konzept	ja	nein	ja
Gerätetypen	ja	optional	irrelevant
Diensttypen	ja	optional	ja
Suche (*Search, Probe*)			
Suche nach Gerätetypen	ja	ja	irrelevant
Suche nach Diensttypen	ja	ja	ja
Suche nach Geräteinstanzen	ja	nein	irrelevant
Suche nach Dienstinstanzen	nein	nein	ja
Laufende Anfragen	nein	nein	ja
Dynamik			
Verhalten bei Änderungen	nicht definiert	repetitiv	iterativ
Protokoll	-	WS-Discovery	DNS-UPDATE, DNS-UL
Discovery außerhalb des lokalen Netzwerkes			
Infrastruktur	-	Discovery Proxy	DNS-Server
Protokoll	-	WS-Discovery	DNS-LLQ

	UPnP	DPWS	WODA
Reduzierung der Netzwerklast			
Verfahren	keine	Suppression, (Discovery Proxy)	Known Answer Suppression, Duplicate Question Suppression, Duplicate Answer Suppression
Praxis			
Einsatz	mangelhaft	nicht erprobt	etabliert
Software	integriert	integriert	separat

instanzen. UPnP fehlt diese Möglichkeit, bietet allerdings äquivalent eine Suche nach einem bestimmten *Root Device* an. In DPWS fehlen beide Möglichkeiten. Treten Änderungen in den Metadaten eines Dienstes oder Gerätes auf, führt DPWS ein vollständig neues Discovery durch (*Bye* und *Hello*). Der Ansatz in WODA ist ein iterativer: Die entsprechenden DNS-Einträge werden einfach für alle sichtbar aktualisiert [143, 144]. UPnP sieht kein Verhalten für sich dynamisch ändernde Parameter vor.

WODA unterstützt zudem als einziges Framework fortlaufende Anfragen (*long-lived queries*, DNS-LLQ). Dieser spezielle, internetweit funktionierende Mechanismus erlaubt eine Suche, die eine definierte Zeit lang gültig bleibt. Ändert sich während dieser Zeit die Antwort, wird sie an den Anfragenden gesendet [145].

Das Discovery außerhalb des globalen Netzwerkes wird durch DNS-Server realisiert, in DPWS mittels Discovery-Proxies. UPnP verfügt über keine derartige Infrastrukturkomponente.

Um die Netzwerklast beim Discovery zu senken bzw. ganz zu umgehen, spezifiziert DPWS einen Suppression-Mechanismus, der ein netzwerkschonendes Verhalten für Clients festlegt, wenn ein Discovery-Proxy im Netzwerk vorhanden ist. Alle zukünftigen Anfragen dürfen dann nur noch an diesen gestellt werden. Zeroconf verfügt über drei derartige Mechanismen, die den Datenverkehr teilweise sogar komplett aufheben können. UPnP sieht kein äquivalentes Konzept vor. Die Erprobung von WS-Discovery steht in der Praxis noch aus. Zeroconf ist bereits in vielen Produkten integriert und es existieren separate, auch frei erhältliche Implementierungen.

Der Ereignismechanismus von WODA ist in seiner Basisversion hinsichtlich seiner Funktionalität sehr schlank gehalten und unterstützt aufgrund seiner Generik eine Vielzahl von Anwendungsfällen. Beim direkten Vergleich mit UPnP und DPWS fallen daher mehrere Unterschiede auf (Tabelle 6.3). Ein Lease-Konzept, welches die Gültigkeit auf eine bestimmte Zeit beschränkt, wird auf der Basisebene nicht vorausgesetzt, wodurch die Implementierungskomplexität sowohl

Tabelle 6.3: Vergleich der Frameworks bezüglich Ereignisse

	UPnP	DPWS	WODA
Allgemein			
Protokoll	GENA	WS-Eventing	HTTP
Abonnement			
Lease-Konzept	ja	ja	nicht definiert
sichtbar	nein	ja	ja
aktualisierbar	nein	nein	ja
Beendigung			
durch Abonnenten	ja	ja	ja
durch Dienst	nein	ja	ja
Reduzierung der Netzwerklast			
Sendefilter	nein	ja	nicht definiert
Datenfilter	nein	nicht definiert	nicht definiert
Mechanismus zur Ereignisübertragung			
Push	ja	ja	ja
Pull	ja	nicht definiert	ja
Architektur			
Vermittler auf Dienstseite möglich	nein	ja	ja

auf Dienstnutzer- als auch auf Dienstseite reduziert wird, da keine Timeouts behandelt werden müssen. Es sei jedoch angemerkt, dass sowohl Dienst als auch Dienstnutzer das Abonnement beenden können (wie auch bei DPWS).

WODA setzt auf der Basisebene eine wichtige Voraussetzung für Erweiterungen: Abonnements sind grundsätzlich sichtbar (einsehbar bzw. abfragbar) sowie aktualisierbar. Beide Eigenschaften resultieren aus der Tatsache, dass Abonnements Ressourcen sind. Definiert ein Domänenprotokoll beispielsweise ein spezifisches Nachrichtenformat für Abonnements, können über den oben gezeigten Mechanismus die Eigenschaften des Abonnements dynamisch abgefragt und geändert werden, wodurch sich Filter oder Leases aktualisieren lassen. DPWS kennt zwar mit *GetStatus* eine solche Abfrage, aber keine Aktualisierung (Ausnahme: *Renew* für Lease-Aktualisierung). In UPnP sind Abonnements versteckt und können nicht geändert werden, weshalb UPnP hier am schlechtesten abschneidet.

Das Abonnieren eines UPnP-Dienstes impliziert die Beobachtung *aller* Status-Variablen, die der Dienst anbietet. Es ist weder die Angabe eines Sende- noch eines Datenfilters möglich. Sendefilter steht hier für einen Mechanismus, der entscheidet, *ob* eine Ereignisnachricht überhaupt geschickt wird oder nicht. Das führt auf der einen Seite zu einer potenziell höheren Netzwerklast, die Implementierung auf Dienstseite ist jedoch relativ leicht, da keine zusätzlichen Berechnungen und Auswertungen durchgeführt werden müssen. DPWS ist hier sehr flexibel gehalten, da zusätzliche Filtermechanismen definiert werden können und mit dem *Action Filter* ein leicht umzusetzender Filter definiert ist. WODA ist auf Basisebene zunächst vergleichbar mit UPnP, da keine Filter definiert werden, jedoch erlaubt der dienstspezifische Schnittstellenentwurf eine sinnvolle Einteilung in abonnierbare Ressourcen unterschiedlicher Granularität. Je nach der hinter einer Ressource stehenden Semantik kann der Ressourcenentwurf daher bereits als Datenfilter aufgefasst werden. Zusätzlich steht mit der Anwendung einer *Resource Factory* ein Pattern zur Verfügung, mit welchem dynamisch neue dienstnutzerspezifische abonnierbare Ressourcen erzeugt werden können. Ein vergleichbares Konzept fehlt sowohl UPnP als auch DPWS.

Die Nachrichtenübertragung kann in UPnP und WODA mittels *Push* oder *Pull* erfolgen. DPWS definiert lediglich *Push*.

Schließlich können mit DPWS und WODA im Gegensatz zu UPnP flexiblere Architekturen umgesetzt werden, in denen Konsumenten, Konsumentvermittler, Produzenten und Produzentvermittler durch unterschiedliche Komponenten realisiert werden, siehe auch Abschnitt 2.3.7.

6.10 Zusammenfassung

Von der Erkenntnis ausgehend, dass das WWW mit seinen etablierten Standards wie URI, DNS und HTTP selbst eine universelle Plattform darstellt, wurde mit der Web-oriented Device Architecture (WODA) konzeptionell ein eigener, alternativer Ansatz für ein Integrationsframework präsentiert. Für das REST-konforme Framework WODA wurden dazu Lösungen entwickelt, die die Grundprinzipien einer serviceorientierten Gerätearchitektur umsetzen.

Im Vergleich mit den Konkurrenztechnologien DPWS und UPnP bietet WODA mit dem Einsatz von Zeroconf besondere Vorteile beim lokalen und internetweiten Discovery von Diensten. Zeroconf ist bereits in vielen Produkten integriert, und es sind einige, u. a. freie, Implementierungen verfügbar. In WODA existieren Geräte nur noch als Gerätebeschreibung. Im Gegensatz zu UPnP und DPWS wird ein Gerät ausschließlich über seine Dienstinstanzen identifiziert. Dieser Ansatz ist sinnvoll, da Dienstnutzer entweder Dienste eines bestimmten Typs oder aber eine bestimmte Dienstinstanz suchen und benutzen.

Der HTTP-basierte Ereignismechanismus ist für die Basisebene einfach umzusetzen, da er auf besondere Funktionalitäten wie Filter- und Lease-Mechanismen verzichtet. Im Gegensatz zu UPnP erlaubt er die Implementierung der beiden Ereignismodelle Peer-to-Peer und Vermittler. Außerdem lassen sich zu überwachende Ressourcen sowohl vom Dienst definieren (während des Ressourcenentwurfes) als auch vom Dienstnutzer bei der Verwendung der vorgestellten *Resource Factory*.

Durch den Einsatz der Ajax-Technologie wird eine ressourcenschonende Nutzung von Geräten durch herkömmliche Webbrowser ermöglicht. WODA benötigt im Gegensatz zu UPnP und DPWS keine zusätzliche Schnittstelle für die Präsentation, sondern kann die Anwendungsressourcen der Dienstimplementierung wiederverwenden. WODA-Dienste benötigen daher nur *eine* Schnittstelle unabhängig von der Art des Dienstnutzers.

7 Die Pipes-Plattform

Die im Abschnitt 1.1 aufgeführten Trends und Folgeerscheinungen führen zu einer weiteren, bisher noch nicht betrachteten Herausforderung: Die alleinige Existenz von (passiven) Diensten reicht allein nicht; es werden Anwendungen benötigt, die die Dienste und Geräte verwenden. In der Regel werden Anwendungen, die Dienste nutzen, in einer bestimmten Programmiersprache implementiert und wie herkömmliche Anwendungen benutzt. Mit der steigenden Anzahl verfügbarer Geräte und Dienste steigen jedoch auch die Möglichkeiten, diese zu kombinieren und neue benutzerspezifische Anwendungen zu schaffen. Die dafür existierenden Möglichkeiten sind für weniger technisch versierte Benutzer zur Zeit sehr begrenzt bzw. nicht vorhanden.

Einen sehr guten Quervergleich bietet die Anwendbarkeit von *Microsoft Excel*. Hier können auch weniger programmiertechnisch erfahrene Benutzer auf einfache Weise „Anwendungen" erstellen. Beispielsweise lassen sich relativ leicht tabellarische oder grafische Auswertungen in Abhängigkeit von erfassten Daten erstellen. Daneben können Formulare entworfen und in die Anwendung integriert werden. Das Aufzeichnen und Abspielen von Makros erlaubt ebenfalls eine einfache und programmierfreie Anwendungserstellung.

Im Abschnitt 3.3 wurden einige existierende grafische Werkzeuge vorgestellt, mit denen sich ebenfalls Anwendungen oder Prozesse erstellen lassen. Sie sind jedoch alle für eine bestimmte Domäne entworfen worden: LabVIEW für Laboranwendungen und Simulationen, BPEL-Designer für die Komposition von Web Services, Yahoo Pipes für die Aggregation von Webdaten.

Der Bedarf, derartige Werkzeuge auch für die Vernetzung von Geräten und Diensten einzusetzen, wird zunehmend größer. Es gibt gegenwärtig kein solches Werkzeug, welches die Verwendung von Geräten und Diensten unterschiedlicher Domänen adressiert, die Integration von Webdiensten ermöglicht und dabei offen und erweiterbar für neue Funktionalitäten ist. Mit der Pipes-Plattform wird gezeigt, wie diese Anforderungen in einem Werkzeug umgesetzt werden können.

7.1 Anforderungen

Im Folgenden werden die Anforderungen und Ziele definiert, die an die Pipes-Plattform gestellt werden:

- **Heterogene, geräteintegrierende Frameworks** Während Werkzeuge für spezifische Frameworks bereits existieren (z. B. BPEL-Designer), ist die Realität durch heterogene

Technologien, Geräte und Dienste gekennzeichnet. Das Werkzeug muss daher framework-agnostisch funktionieren, also für UPnP, DPWS und WODA gleichermaßen verwendet werden können. Für einen Benutzer ist das konkrete Framework ohnehin zweitrangig.

- **Domänen** Pipes soll in verschiedenen Domänen eingesetzt werden können. Ziele sind die Gebäudeautomatisierung, die Anwendung im Desktopbereich, im Internet sowie auf eingebetteten Systemen.
- **Zielgruppen** Die Plattform soll mindestens zwischen den Anbietern von Funktionalitäten und den Anwendern auf Designebene unterscheiden und beiden das unabhängige Entwickeln ermöglichen.
- **Homogenität** Die von der Plattform zur Verfügung gestellten Funktionalitäten sollen in einer technisch homogenen Art und Weise abgebildet werden, um die Bedienung zu vereinfachen.
- **Einfachheit** Weniger technisch versierte Benutzer sollen Prozesse, Anwendungen oder Dienste definieren und ausführen können. Unter anderem folgt daraus, dass keine Programmierkenntnisse vorausgesetzt werden dürfen.
- **Erweiterbarkeit** Das Werkzeug muss leicht erweiterbar sein, so dass technisch versierte Entwickler neue Funktionalitäten implementieren und Benutzern anbieten können.

7.2 Grundidee und Konzept

Die konzeptionelle Grundidee für die Architektur zeigt Abbildung 7.1. Sie besteht aus drei Schichten und unterscheidet zwischen drei verschiedenen Benutzern. Auf der unteren Ebene befinden sich domänen- und technologiespezifische Bibliotheken und Programme. Diese sind verantwortlich für die konkrete Umsetzung der entsprechenden Funktionalitäten, z. B. für DPWS, Google Maps oder XML. Die Bereitstellung erfolgt durch Bibliotheksentwickler.

Auf der mittleren Ebene befinden sich Module. Während die Bibliotheken unabhängig von der Plattform existieren können, sind Module bereits Konzepte der Plattform. Sie stehen jeweils für eine spezifische Aufgabe und verwenden die Bibliotheken. Sie werden von Modul-Entwicklern implementiert. Daher müssen Modul-Entwickler sowohl mit der Plattform als auch mit den entsprechenden Bibliotheken umgehen können.

Die obere Ebene stellt die Anwendungsdesignebene dar. Hier werden Module grafisch miteinander verbunden, wodurch Pipes bzw. Anwendungen entstehen. Die Modulimplementierungen, und damit die technische Sicht, sind vor den Modul-Entwicklern verborgen. Die Pipes-Plattform verfolgt mit den miteinander verbundenen Modulen auf grafischer Ebene den gleichen Designansatz wie LabVIEW, Yahoo Pipes oder die OutSystems Plattform.

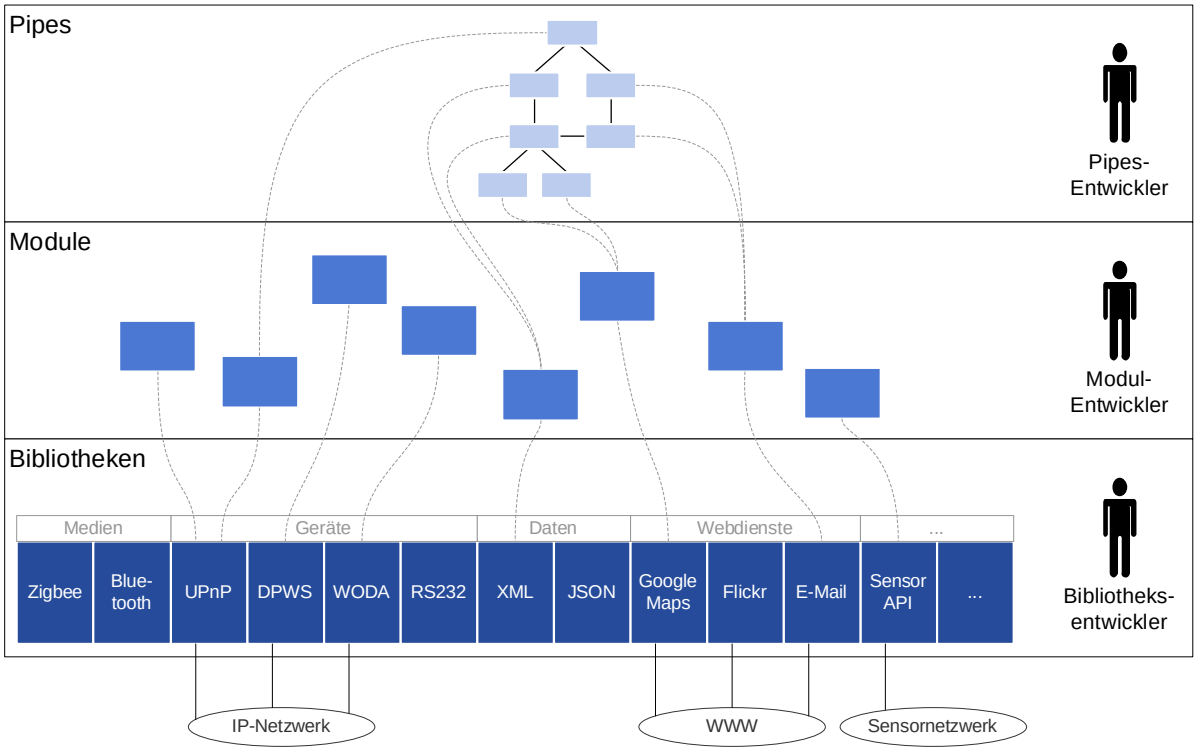

Abbildung 7.1: Modulares Konzept für die Umsetzung der Anforderungen an die Pipes-Plattform

Hauptanwendung der Pipes-Plattform ist ein in einem Webbrowser lauffähiges, grafisches Modellierungswerkzeug – der *Pipes-Modeler* – mit dem neue Dienste, Anwendungen oder Prozesse (*Pipes*) erstellt werden können. Die so erstellten Pipes werden von der *Pipes-Engine* interpretiert und ausgeführt. Mit dem *Monitor* lassen sich in Ausführung befindliche Pipes überwachen. Die Pipes-Plattform basiert auf einem dynamischen Plug-in-Konzept und unterstützt dadurch die Integration neuer Funktionalitäten durch Drittanbieter.

Die Pipes-Plattform verwendet die OSGi-Plattform (Abs. 3.1). Über diese lassen sich heterogene Frameworks benutzen, da bereits entsprechende Implementierungen für OSGi zur Verfügung stehen. OSGi zielt außerdem auf eingebettete Systeme ab, weshalb Pipes auch dort eingesetzt werden kann. Die Modellierung erfolgt in einem herkömmlichen Webbrowser und ist damit nicht an den Ort der Plattform gebunden. Mit der oben dargestellten Schichtenarchitektur und der Verwendung von OSGi als zentrale Ausführungsumgebung (Abbildung 7.2) erfüllt die Pipes-Plattform die genannten Anforderungen. In den nächsten Abschnitten werden weitergehende Konzepte sowie die Umsetzung erörtert.

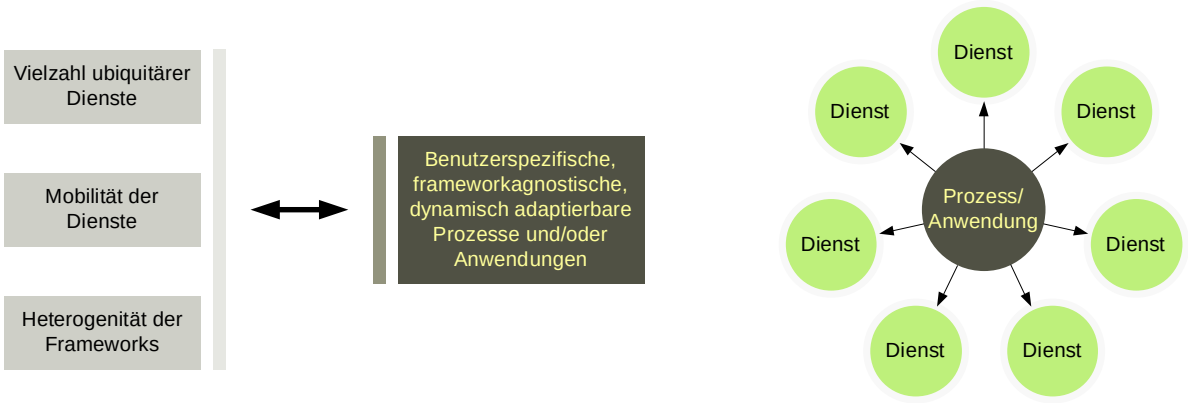

Abbildung 7.2: Zentraler Steuerungsansatz für flexible Prozesse und Anwendungen unter Berücksichtigung der Anforderungen in der Pipes-Plattform

7.2.1 Architektur

Die zentralen Komponenten der Pipes-Plattform sind *Module*, *Modulkonfigurationen*, *Ports*, *Kabel* und *Pipes*. Ein Modul kann über seine In-Ports Daten empfangen und über die Out-Ports Daten senden. Durch Module werden funktionale Komponenten, Dienste und Geräte homogen abgebildet. Die nachfolgenden Beispiele zeigen typische Anwendungsfälle für Module, das Modulkonzept ist jedoch nicht auf sie begrenzt. Vielmehr lässt sich alles als Modul abbilden, was in irgendeiner Weise Daten erzeugen und/oder verarbeiten kann:

- Ein konkretes Gerät wie eine Webcam eines bestimmten Modells, die über eine spezifische IP-Adresse erreichbar ist
- Ein Gerätetyp wie ein UPnP-AV-Media-Renderer, welcher über eine API oder ein spezielles Geräteprotokoll angesprochen werden kann
- Ein Signaturdienst, der Teile einer XML-Nachricht verschlüsselt
- Ein DPWS-Discovery-Dienst, der an- und abmeldende Geräte in einem lokalen Netz bekannt gibt
- Ein Konvertierungsdienst, der aus einer CSV-Liste (*Comma Separated Value*) eine XML-Liste erzeugt
- Ein Dienst, der Google-Maps-Karten erzeugt
- Ein Dienst, der Daten einer Datenbank zur Verfügung stellt
- Ein Dienst, der SMS-Nachrichten verschickt
- usw.

Damit überhaupt ein Datenverkehr zwischen Modulen stattfinden kann, müssen die Ports über Kabel miteinander verbunden werden. Das geschieht innerhalb einer Pipe. Mit einer Pipe kann ein Prozess oder eine Anwendung erstellt werden, indem Module in geeigneter Weise miteinander kombiniert werden. Eine solche Pipe besteht aus einer bestimmten Anzahl von Modulen, wobei gleiche Module auch mehrfach verwendet werden können. Diese können modulspezifisch konfiguriert und miteinander über ihre Ports „verkabelt“ werden.

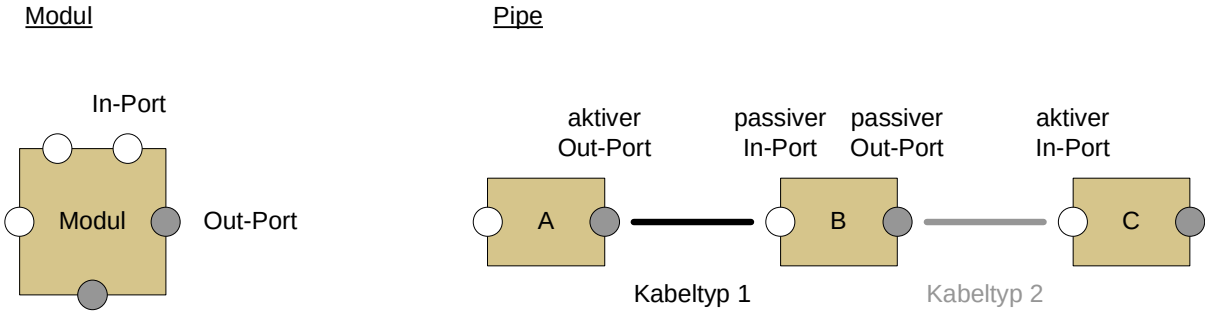

Abbildung 7.3: Module, Ports, Kabel und Pipes

Ein Kabel verbindet grundsätzlich einen In-Port mit einem Out-Port (Abbildung 7.3). Es gibt zwei Typen von Kabel, die damit die Ports, die sie miteinander verbinden, konfigurieren. Durch Kabeltyp 1 wird der Out-Port aktiv und der In-Port passiv. Bei dieser Variante muss das Modul mit dem In-Port auf Daten warten. Vom anderen Modul wird erwartet, dass es selbstständig Daten sendet. Der Kabeltyp 2 kehrt diese Konfiguration um. Vom Modul mit dem nun aktiven In-Port wird erwartet, dass es über diesen Daten anfragt. Das andere Modul wartet an seinem passiven Out-Port auf eine Anfrage und darf erst dann Daten senden. Durch diese Unterscheidung kann eine bessere Flusssteuerung erreicht werden als bei Verwendung eines einzigen Kabeltyps. Sie wird später in einem Beispiel erläutert. Yahoo Pipes, Macro.scopia usw. kennen dieses Konzept nicht.

Module kapseln ihre Daten und Implementierungen und sind vor äußeren Zugriffen durch andere Module geschützt. Diese Eigenschaft ist symmetrisch und gilt sowohl für die innere als auch die äußere Modulsicht gleichermaßen: Eine Modulimplementierung kommuniziert ausschließlich mit den Ports ihres Moduls, indem es über diese Daten empfängt und versendet. Sie kennt weder die Module, die mit ihr kommunizieren, noch den Aufbau und das Ziel der Pipe, von der sie verwendet wird. Die Modulimplementierung kann für ihre Ausführung also keine bestimmten Annahmen voraussetzen und ist demnach ausschließlich auf die Modulkonfiguration und die Daten angewiesen, die sie über ihre Ports erhält. Umgekehrt können Module nicht direkt mit

einer bestimmten Modulimplementierung kommunizieren, sondern lediglich indirekt, indem sie Daten an ihre Ports senden oder von dort empfangen.

Das Verhalten eines Moduls wird allein durch dessen Implementierung definiert. Die Pipes-Plattform erteilt, mit Ausnahme der Semantik der beiden Kabeltypen, keine weiteren Auflagen darüber, wie oft und unter welchen Umständen ein Modul Daten annehmen oder versenden muss. Dieses Verhalten ist informell in einem *Modulvertrag* geregelt. Dieser muss einem Nutzer der Plattform zur Verfügung gestellt werden, damit er das Modul entsprechend korrekt einsetzen kann.

Die Pipes-Plattform verfolgt damit den Ansatz der *autonomen Komponenten*, so wie er auch bei Bauteilen integrierter Schaltkreise verwendet wird (im Hardware-Sprachgebrauch: *Black Boxes*). Eine ähnliche Architektur ist in JSim [146, 147], einem Netzwerksimulator, zu finden. Dort implementieren *Komponenten* Netzwerkprotokolle. Anschließend können diese über *Ports* miteinander verkabelt werden, wodurch eine Simulationsanordnung entsteht. Die Komponenten von JSim verfolgen ebenfalls den Ansatz autonomer Komponenten, jedoch ist JSim ein reiner Netzwerksimulator [148] und verfolgt nicht die Ziele der Pipes-Plattform.

7.2.2 Ausführung

Eine Pipe ist gekennzeichnet durch eine Menge von Modulen, Modulkonfigurationen und Kabeln, die die Module verbinden. Hierbei handelt es sich um reine Metadaten. Daher können Pipes auf einem beliebigen Interpreter ausgeführt werden, vorausgesetzt, dieser hat Zugriff auf die erforderlichen Module.

Die Hauptaufgabe des Interpreters liegt in der Vermittlung von Daten. Wenn ein Interpreter eine Pipe ausführt, muss er zunächst die entsprechenden Modulinstanzen erzeugen und alle Module *gleichzeitig* starten. Aufgrund der Autonomie der Module kann der Interpreter keine Annahmen über die Aufgabe der Pipe treffen und nicht wissen, welche Module zuerst Daten senden wollen. Anschließend muss der Interpreter Daten von Modulen entgegennehmen und an die entsprechenden angeschlossenen Module weiterreichen. Der Interpreter agiert demnach als globaler Nachrichtenverteiler. Werden zahlreiche Module eingesetzt, ist der Interpreter für eine homogene Nachrichtenverteilung verantwortlich, um Wartezeiten zu vermeiden. Weiterhin muss er mit dem Pipes-Monitor zusammenarbeiten, mit welchem Prozesse gestartet, angehalten und beendet werden können.

7.3 Umsetzung

7.3.1 Architektur der Plattformsoftware

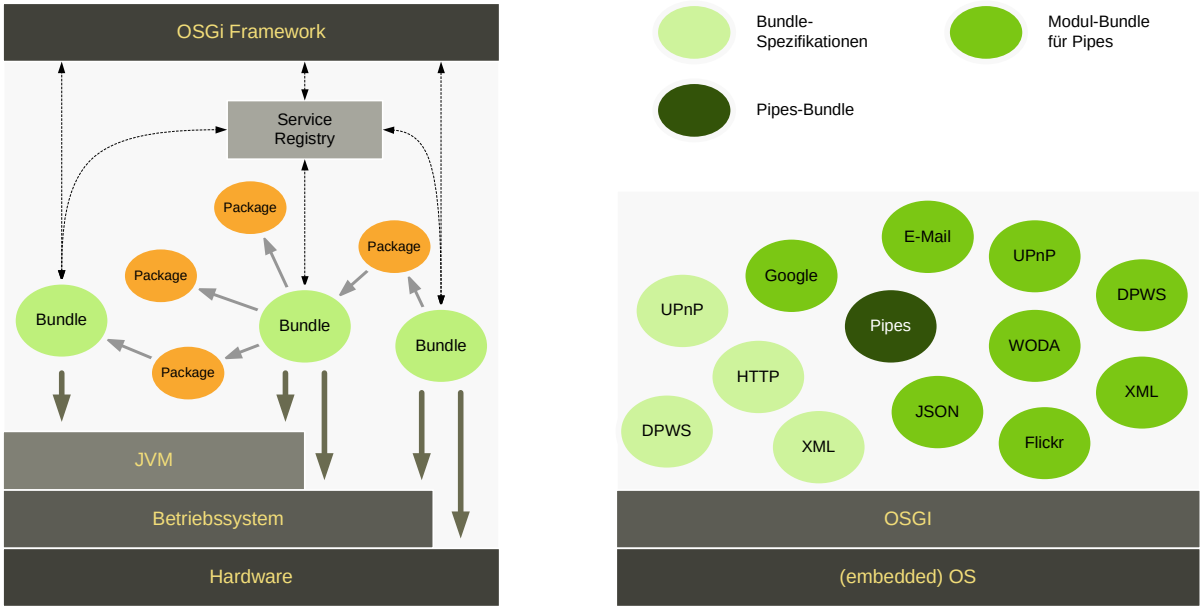

Abbildung 7.4: Module und Pipes als OSGi-Bundles

Die Pipes-Plattform setzt vollständig auf die OSGi-Technologie auf. OSGi erweitert, wie im Abschnitt 3.1 beschrieben, Java um dynamische Aspekte. OSGi erreicht das, indem sämtliche Anwendungen (Bundles) in der gleichen Java Virtual Machine laufen. Die Bundles sind voneinander getrennt und nur über vordefinierte Interfaces ansprechbar. Bundles können zur Laufzeit Dienste exportieren, die von anderen Bundles entdeckt und benutzt werden können (Plug-and-Play). Auf diese Art und Weise werden Java-Anwendungen zu Diensten, für welche OSGi die dazu notwendige SOA-Infrastruktur bereitstellt (Abbildung 7.4).

Eine Anforderung an die Pipes-Plattform war die Möglichkeit, diese mittels eines Plug-in-Konzeptes um Module zu erweitern, eine andere, dass möglichst einfach Geräte bzw. existierende Technologien in das Gesamtkonzept involviert werden können. Die Wahl fiel auf OSGi, da es mit dem dynamischen Installieren und Deinstallieren von Bundles (Modulen) zur Laufzeit bereits einen Teil der Plug-in-Funktionalität abdeckt. Außerdem bringt es bereits UPnP-, DPWS-, HTTP-Server- und andere Bundles mit, die die entsprechenden Technologien als Interfaces abbilden und somit mitgenutzt werden können. Im Falle von UPnP und HTTP sind diese bereits vom OSGi-Konsortium standardisiert. Dadurch ist die Interoperabilität mit unterschied-

lichen OSGi-Plattformen und Bundle-Implementierungen gewährleistet. Die Standardisierung des DPWS-Treibers [149, 150] ist derzeit Gegenstand des AMIGO-Forschungsprojektes [151].

An dieser Stelle zahlt sich aus, dass OSGi ursprünglich als Plattform für eingebettete Systeme entwickelt wurde, die verschiedene Geräteprotokolle miteinander verbinden sollte. Der Fokus von OSGi hat sich inzwischen zwar vergrößert, jedoch wird es nach wie vor beispielsweise im Automobilbau oder in der Energiewirtschaft eingesetzt.

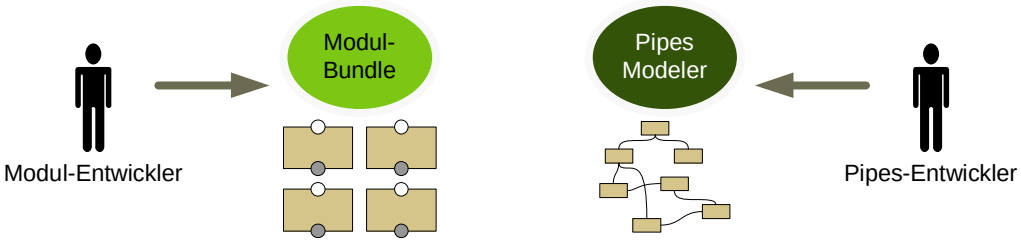

Abbildung 7.5: Technische (Modul-Entwickler) und semi-technische (Pipes-Entwickler) Benutzer der Pipes-Plattform

Die Pipes-Plattform besteht aus einem Pipes-Bundle und mehreren optionalen Modul-Bundles. Das Pipes-Bundle enthält den Pipes-Modeler, die Pipes-Engine und den Monitor. Der Pipes-Modeler wird von Pipes-Entwicklern verwendet (Abbildung 7.5). Modul-Bundles enthalten die Modulimplementierungen. Es ist beabsichtigt, dass diese von Dritten (Modul-Entwicklern) entwickelt und als herunterladbare Bundles im Internet zur Verfügung gestellt werden.

7.3.2 Pipes-Modeler

Der *Pipes-Modeler* ist das zentrale Werkzeug, um Pipes zu erzeugen, zu bearbeiten und auszuführen. Der Pipes-Modeler ist eine Webanwendung und kann in einem herkömmlichen Webbrowser verwendet werden. Er bildet die grafische Schnittstelle zwischen der Pipes-Plattform und dem Benutzer (Abbildung 7.6).

Die Oberfläche teilt sich in zwei Bereiche auf, die jeweils eine Ansichtskomponente darstellen können. Zwischen den Ansichtskomponenten kann mit Tabs (Karteireiterprinzip) umgeschaltet werden. In der *Pipesansicht* sind alle auf der Plattform abgelegten Pipes aufgeführt. Von hier aus können die Pipes in die *Modellieransicht* geladen werden, wobei für jedes geöffnete Pipes-Dokument auch eine Modellieransicht zur Verfügung steht. Dadurch lassen sich zeitgleich mehrere Pipes bearbeiten. Die Modellieransicht enthält weiterhin Optionen für das Speichern, Kopieren, Exportieren und Löschen von Pipes. Außerdem lassen sich von ihr aus die Pipes starten.

Die *Modulansicht* listet alle derzeit auf der Plattform zur Verfügung stehenden Module alpha-

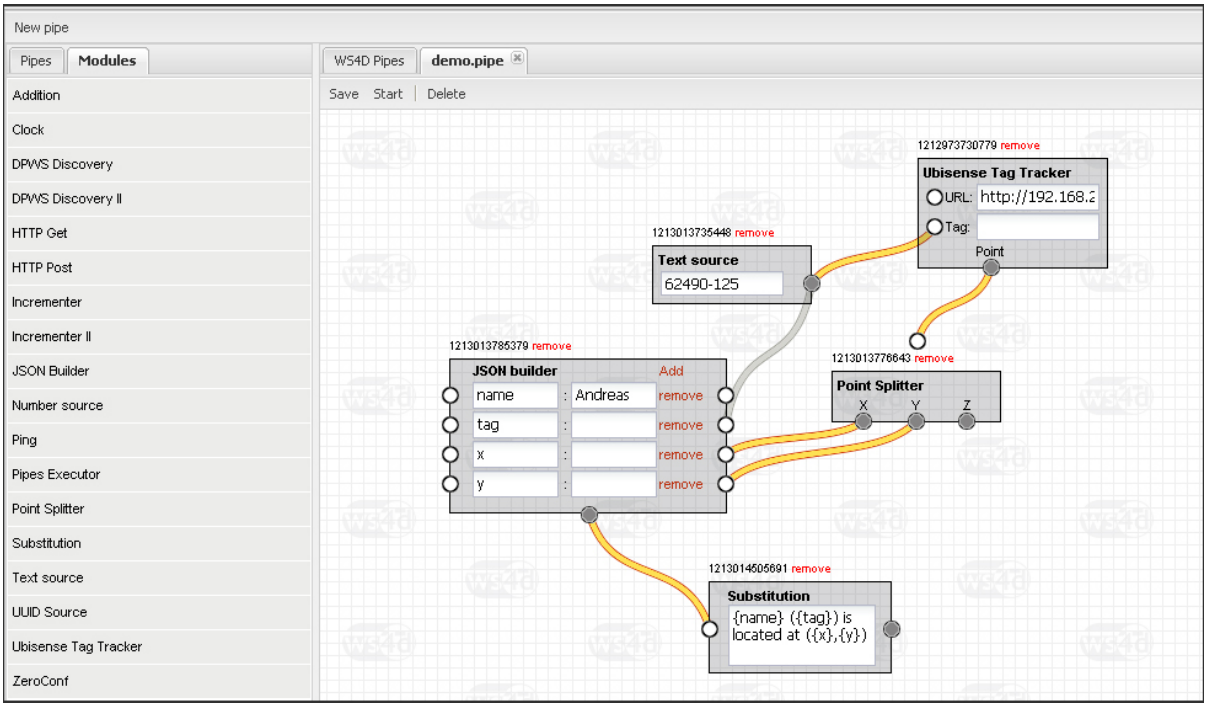

Abbildung 7.6: Screenshot des Pipes-Modelers

betisch auf. Um ein Modul zu verwenden, wird es mittels Drag-and-Drop in die Modellieransicht gezogen. Die Darstellung der Module ist modulspezifisch und vom Modul-Entwickler frei gestaltbar. Da die Plattform lediglich das Modulprinzip vorgibt, jedoch nicht das Anwendungsfeld einzelner Module vorherbestimmt oder einschränkt, wurde auch hier bei der visuellen Darstellung das Plug-in-Konzept der gesamten Pipes-Plattform konsequent fortgesetzt. Es können alle Möglichkeiten, die HTML, CSS und JavaScript bieten, ausgenutzt werden. Dadurch lassen sich Elemente wie Tooltips, Bilder, Hovers usw. verwenden, wodurch die Modulrepräsentationen entsprechend intuitiv gestaltet werden können. Mittels JavaScript lassen sich selbst komplexe Wizards und Dialoge abbilden, um beispielsweise Module zu konfigurieren. Diese Möglichkeit der freien Gestaltbarkeit der Benutzerschnittstelle von Modulen kann als erheblicher Vorteil gegenüber anderen Plattformen gewertet werden. Als Entwurfsrichtlinie ist jedoch empfohlen, funktional gleichartige Module (beispielweise alle Web-Services-Module) in einem durchgängigen optischen Stil anzubieten.

Die Darstellung einiger weniger Elemente wird durch die Plattform vorgegeben. So ist über jedem Modul ein eindeutiger Bezeichner (Modul-ID) angegeben. Dadurch lassen sich beispielsweise beim Monitoring typgleiche Module voneinander unterscheiden. Ebenso ist die Darstellung

von In- und Out-Ports für alle Module gleich gehalten. Module brauchen sich daher nicht um die Darstellung von Ports zu kümmern, sondern müssen dem Modeler lediglich mitteilen, *wo* diese platziert werden sollen und welchen Typ sie besitzen sollen. Module werden durch Ziehen mit der Maus von einem zum anderen Port miteinander verbunden. Dabei wird jeweils ein Kabel erzeugt. Der Kabeltyp lässt sich mit einem Klick umschalten.

In der *Monitoringansicht* werden alle gestarteten Pipes (im folgenden *Prozesse*) aufgelistet. Von hier aus können sie angehalten, fortgesetzt, beendet und gelöscht werden. Die Module erzeugen während ihrer Ausführung Nachrichten, die hier ebenfalls eingesehen werden können. Ein und dieselbe Pipe kann mehrere Male gestartet werden, wobei bei jedem Start ein neuer Prozess angelegt wird.

7.3.3 Plug-in von Modulen

Module werden durch *Modul-Bundles* der Plattform zur Verfügung gestellt. Hierbei handelt es sich um herkömmliche OSGi-Bundles (Abbildung 7.4). Daher können sie zur Laufzeit sowohl installiert als auch wieder entfernt werden. Ein Modul-Bundle muss einen OSGi *BundleActivator* enthalten, der einen *ModuleProvider* als OSGi-Service bei der OSGi-Plattform registriert. Auf der anderen Seite überwacht das *Pipes-Bundle* das OSGi-Framework auf neue *ModuleProvider*.

Um neue Module zu implementieren, stellt die Pipes-Plattform zwei Plug-in-Mechanismen zur Verfügung: Für die serverseitige Modulimplementierung müssen Module das *Module*- und das *ModuleInstance*-Java-Interface implementieren, für die Clientanwendung im Pipes-Modeler müssen sie vordefinierte JavaScript-Methoden anbieten. JavaScript als dynamische Sprache kennt das Konzept von Interfaces nicht. Deshalb kann die Schnittstelle nur informal angegeben werden.

Demnach besteht eine Moduleinheit mindestens aus einer oder mehreren Java-Klassen, die das *Module*- und das *ModuleInstance*-Interface implementieren und einer JavaScript-Datei, die auf Clientseite mit dem Pipes-Modeler kommuniziert. Des Weiteren kann eine Moduleinheit CSS-, HTML-Dateien sowie weitere Ressourcen wie Bilder mitbringen, die für die Darstellung im Webbrowser benötigt werden.

Listing 7.1 zeigt auszugsweise das Grundgerüst einer JavaScript-Datei, die ein Modul (hier: *JSON-Builder*) implementieren muss. Es besteht aus der Definition einer Konstruktorfunktion und seiner Registrierung beim Modulmanager des Pipes-Modelers (Zeile 29). Die Registrierung erfolgt unter Angabe eines Typen. Jedes Modul besitzt einen Typ, der es identifiziert. Der Modultyp findet in der Plattform durchgehend Anwendung – im Monitoring, beim Öffnen und Speichern für das Dokumentenformat, beim Instanziieren von Modulen für Prozesse sowie im Pipes-Modeler für das Rendering. Da der Modultyp auch plattformübergreifend Bestand haben

Listing 7.1: Auszug aus einer Modul-JavaScript-Datei am Beispiel des JSON-Builders

```
Pipes.mod.json.Builder = function(context) {

    var state;

    this.render = function(wrapper) {
        ...
    }

    this.setState = function(o) {
        state = o;
    }

    this.getState = function() {
        var o = {
            keyValues : []
        };
        keys.eachKey(function(key) {
            o.keyValues.push({
                id : key,
                keyString : keys.get(key).getValue(),
                valueString : values.get(key).getValue()
            })
        });
        return (o);
    }

}

Pipes.ModuleMgr.registerType('json.builder', Pipes.mod.json.Builder);
```

sollte, um Kollisionen zu verhindern, kommt der Wahl eines eindeutigen Typbezeichners eine wichtige Bedeutung zu. Als Empfehlung gilt, die Punktnotation zu verwenden, beispielsweise *http.delete*, *dpws.discovery.proxy* oder *util.text.filter*, da sich dadurch leicht logische Hierarchien abbilden lassen. Diese sind leicht lesbar und verständlich und haben sich bereits bewährt (z. B. in Java, C# usw.).

Die Konstruktorfunktion wird vom Pipes-Modeler immer dann aufgerufen, wenn ein entsprechendes Modul erzeugt werden soll. Das ist nach einem Drag-and-Drop sowie beim Öffnen eines Pipe-Dokumentes der Fall. Die Funktion muss folgende Methoden anbieten:

- *getState(): Object* Module können im Modeler konfiguriert werden. Typische Eingaben werden durch herkömmliche HTML-Formulare realisiert oder können von komplexen Wizards vorgenommen werden. Da die dadurch erzeugten Einstellungen abhängig vom jeweiligen Modul sind, musste ein Weg gefunden werden, diese generisch persistent abzulegen. Als einfache und natürliche Schnittstelle eignet sich dazu JSON, da sie sowohl in JavaScript als auch in jeder anderen Programmiersprache als Bibliothek verfügbar ist. Mit JSON las-

sen sich sehr leicht Hierarchien, Schlüssel-Wert-Paare und Listen abbilden. Die *getState*-Methode wird vom Modeler aufgerufen, wenn ein Pipes-Dokument gespeichert werden soll. Das Modul bestimmt Aufbau und Inhalt eines solchen Zustandsobjektes. Es ist immer nur vom jeweiligen Modul interpretierbar.

- *setState(o: Object): void* Nach dem Öffnen eines Pipe-Dokumentes erhält das Modul hier seinen letzten Zustand zurück.
- *render(wrapper: DOMElement): void* Wird vom Modeler aufgerufen, wenn das Modul im Webbrowser dargestellt werden soll. Dazu wird ein HTML-DOM-Element übergeben, welches als Container dient. Bringt die Moduleinheit bereits eine HTML-Datei mit, wird diese hier automatisch eingebettet. Die *render*-Methode wird nach *setState* aufgerufen und muss hier bei der Darstellung den gesetzten Zustand entsprechend berücksichtigen.

Wie in Listing 7.1 ersichtlich, erhält ein Modulkonstruktor ein *ModuleContext*-Objekt, welches als Verbindung zum Modeler dient. Hierüber lassen sich dynamisch Ports zum Modul hinzufügen, positionieren und auch wieder entfernen.

Die Schnittstelle für Module wird vom Pipes-Bundle auf Serverseite vorgegeben und besteht aus nur zwei leichtgewichtigen Java-Interfaces. Das *Module*-Interface definiert folgende Methoden:

- *getType(): String* Hier muss der Typ, welcher auch in der JavaScript-Implementierung verwendet wird, zurückgegeben werden.
- *getLabel(): String* Während der Typ nur intern verwendet wird, werden die Module im Modeler mit einem Label dargestellt, welches das Modul kurz und eindeutig beschreiben soll.
- *getModuleInstance(): ModuleInstance* Eine *ModuleInstance* enthält die eigentliche Implementierung eines Moduls. Sie interagiert aktiv und passiv mit der Pipes-Engine während der Ausführung eines Pipe-Prozesses und hält den aktuellen Modulzustand. Modulinstanzen werden von der Pipes-Engine erzeugt, wenn ein neuer Prozess gestartet wird. Dazu wird diese Methode für jedes modellierte Modul aufgerufen. Wird derselbe Modultyp mehrmals verwendet, wird auch diese Methode mehrmals aufgerufen.

Das *ModuleInstance*-Interface ist als eigenständiges Interface definiert und deklariert die folgenden Methoden:

- *init(e: Executor, state: JSONObject)* Wird von der Engine aufgerufen, nachdem die Instanzen erzeugt, aber noch nicht gestartet wurden. Über den *state*-Parameter erhält die

Modulinstanz die Daten, die das JavaScript im Modeler zurückgegeben hat. In den meisten Fällen hat dieses Zustandsobjekt direkte Auswirkungen auf die Arbeitsweise des Moduls.

- *start()*, *pause()*, *resume()* und *stop()* Diese Methoden sind weitestgehend selbsterklärend und werden von der Engine immer für alle Module zusammen aufgerufen. Wichtig ist anzumerken, dass nach Aufruf der *stop*-Methode die Modulinstanzen alle etwaigen Abhängigkeiten zu anderen Objekten beseitigen müssen, damit sie selbst vom *Java Garbage Collector* entfernt werden können.
- *receiveAtPort(port: String, value: Object, ec: ExecutionContext)* Wenn über ein Kabel Daten an einen passiven In-Port gesendet werden, ruft die Engine im entsprechenden Modul *receiveAtPort* mit dem Namen des Ports und den Daten auf.
- *responseAtPort(port: String, ec: ExecutionContext): Object* Wird aufgerufen, wenn ein passiver Out-Port angefragt wird. Die Methode muss ein Objekt zurückgeben, welches als Antwort für das aufrufende Modul dient.

Die Modul-Interfaces dienen also hauptsächlich zur Steuerung durch die Engine. Umgekehrt muss es jedoch auch eine Möglichkeit für Module geben, mit der Engine zu kommunizieren, etwa dann, wenn Daten aktiv gesendet werden sollen. Dazu stehen zwei weitere Interfaces zur Verfügung, die vom Pipes-Bundle exportiert werden: *Executor* und *ExecutionContext*.

Alle Modulaktivitäten während der Ausführung eines Pipe-Prozesses geschehen innerhalb eines jeweiligen *ExecutionContexts* (EC). ECs werden vom *Executor* verwaltet und kontrolliert an die Module verteilt. Es gibt zwei unterschiedliche Wege, um einen EC zu erhalten: In den meisten Modulmethoden befindet sich bereits ein EC in der Signatur und mittels *getExecutionContext()* kann ein gültiger EC durch den *Executor* erzeugt werden. Damit ein Modul Daten senden kann, benötigt es einen EC. Entsprechend sind u. a. die zwei folgenden wichtigen Methoden im *ExecutionContext* zu finden:

- *sendViaPort(port: String, value: Object)* Muss von einem Modul aufgerufen werden, wenn es Daten über einen aktiven Out-Port senden möchte. Die Methode blockiert nicht und kehrt sofort zurück.
- *requestViaPort(port: String): List<Object>* Wird aufgerufen, um Daten über einen aktiven In-Port anzufragen. Die Methode blockiert, bis alle angeschlossenen Module geantwortet haben und gibt die Ergebnisse als Objektliste zurück.

Die weiteren Methoden im *ExecutionContext* erlauben das Erzeugen von Nachrichten, die während des Monitorings eingesehen werden können. Schließlich bietet der *Executor* noch einige grundlegende Funktionalitäten an. Beispielsweise ist es einem Modul möglich, eine andere Pipe

zu starten und diese auch wieder zu beenden. Dadurch sind Modulimplementierungen möglich, die Schleifen erzeugen.

7.3.4 Die Pipes-Engine

Pipes-Prozesse werden von der *Pipes-Engine* ausgeführt und überwacht. Es können mehrere Prozesse gleichzeitig ausgeführt werden. Jeder Prozess wird dabei von einem *Executor*-Objekt gesteuert. Die Pipes-Engine arbeitet dazu mit dem Pipes-Monitor zusammen, welcher Teil des Pipes-Modelers ist.

Listing 7.2: Auszug aus dem zum Screenshot zugehörigen Pipes-Dokument

```
{
    "modules": [
        {
            "ports": [
                {
                    "name": "feedIn",
                    "type": "in"
                },
                {
                    "name": "resultOut",
                    "type": "out"
                }
            ],
            "name": 1213014505691,
            "state": {"template": "{name} ({tag}) is located at ({x},{y})"},
            "target": 4,
            "type": "substitution",
            "y": 448,
            "x": 572
        }, ...
    ], ...
}
```

Beim Starten wird das entsprechende Pipes-Dokument vom Executor geladen und geparst. Nacheinander werden die Modulinstanzen erzeugt, mit dem jeweiligen im Pipes-Dokument hinterlegten Zustand initialisiert und gestartet. Einen Ausschnitt eines Pipes-Dokumentes zeigt Listing 7.2, Anhang D enthält das vollständige Beispiel.

Der Executor verwaltet in einem Pool *ExecutionContext*-Objekte, die auf Anfrage der Module an diese temporär zugewiesen und wieder zurückgeholt werden. ECs laufen jeweils in einem eigenen Thread. Die maximale Anzahl lässt sich in der OSGi-Konfigurationsdatei einstellen.

Die Module der Pipes-Plattform stellen aus architektonischer Sicht autonome Komponenten dar. Sie kennen weder die anderen Module, mit denen sie kommunizieren, noch kennt der Exe-

cutor die Funktion einzelner Module oder das Ziel eines Pipes-Prozesses. Daher wissen weder Module noch Executor wann ein Prozess *beendet* ist. Eine Ausnahme bilden Module, die selbst Pipes (Subpipes) starten. Pipes-Prozesse laufen unendlich lange, es sei denn sie werden explizit von einem Benutzer der Plattform oder von einem Modul gestoppt oder aber die Ausführung der Plattform wird beendet. Im Pipes-Monitor können Prozesse deshalb angehalten, fortgesetzt und gestoppt werden.

Als direkte Konsequenz aus der Verwendung von Threads können Module keine Annahmen über die zeitliche Zustellung von Daten treffen bzw. eine bestimmte Reihenfolge erzwingen. Schickt ein Modul beispielsweise nacheinander die Daten *D1* und *D2* über einen Out-Port, an welchem zwei Module *A* und *B* angeschlossen sind, so ist lediglich sichergestellt, dass beide Module beide Daten empfangen werden. Es ist jedoch möglich, dass *D2* zeitlich vor *D1* eintrifft oder dass ein Modul bereits beide, das andere Modul aber noch keine Daten empfangen hat. Spielt die Reihenfolge eine Rolle, muss ein entsprechender Mechanismus in den Daten abgebildet werden oder aber die Module müssen ein synchronisiertes Verhalten aufweisen. Ein solches Verhalten wird im nächsten Absatz vorgestellt.

7.3.5 Modul-Design in der Praxis

Die Pipes-Plattform stellt eine sehr generische Umgebung zur Verfügung, die sich neben der Prozessmodellierung für Geräte für eine Vielzahl anderer Anwendungsfälle eignet. Aus diesem Grunde sind die Bedingungen für das Senden und Empfangen von Daten sehr großzügig gehalten.

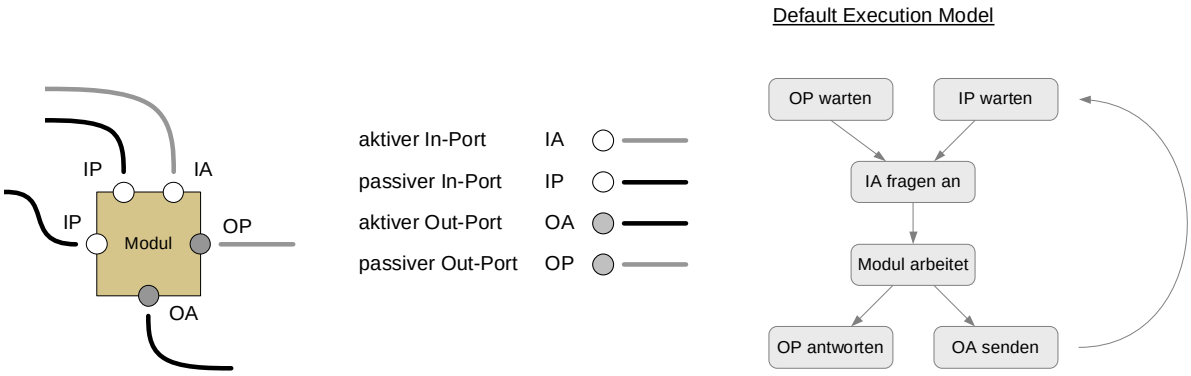

Abbildung 7.7: Typische Implementierung eines Moduls

In der Praxis sind aber nicht ausschließlich Module erwünscht, die in unberechenbarer Weise Daten senden. Daher wird hier das *Default Execution Model* definiert, welches in nahezu allen Fällen zum Einsatz kommen kann. Diese bevorzugte Modulimplementierung arbeitet in einer

feststehenden Reihenfolge seine verkabelten Ports ab und ist daher für Pipes, die sequenzielle Abfolgen verwenden, geeignet (Abbildung 7.7).

Ein Modul, welches das *Default Execution Model* implementiert, weist ein insgesamt passives Verhalten auf. Es wartet zunächst an seinen passiven Ports auf Anfragen (Out-Ports) bzw. Daten (In-Ports). Anschließend fragt es selbst über seine aktiven In-Ports Daten ab. Erst danach beginnt die eigentliche Arbeit des Moduls, wobei in den meisten Fällen Daten aus den empfangenen Daten berechnet werden. Schließlich sendet das Modul die Ergebnisse über die aktiven Out-Ports bzw. beantwortet die Anfragen an den passiven Out-Ports und beginnt die Arbeit wieder von vorne.

Dieses Default-Verhalten zusammen mit der Möglichkeit, Ports aktiv oder passiv zu konfigurieren, erlaubt eine flexible Arbeitsweise eines Moduls in unterschiedlichen Kontexten (Pipes).

Die Pipes-Plattform stellt ebenfalls keine Bedingungen hinsichtlich der Daten, die über die Kabel gesendet werden. In den Interfaces sind sie als *Object* deklariert. Um jedoch eine hohe Interoperabilität zwischen den verteilt entwickelten Modulen zu gewährleisten, wird empfohlen, nur die simplen Typen wie *String*, *Number*, *Boolean* und *XML* oder *JSON* als strukturierte Typen zu verwenden. Spezielle Module zur Datenmanipulation erlauben eine sehr flexible Anwendung dieser Basistypen.

7.3.6 Subpipes

Als *Subpipe* wird eine Pipe bezeichnet, die durch eine andere Pipe (innerhalb eines Moduls) gestartet wird. Subpipes sind selbst nicht Konzept der Plattform, jedoch können diese durch geeignete Module realisiert werden, da der *Executor* das Starten und Beenden beliebiger Pipes zulässt.

7.4 Anwendungsbeispiel

Die Pipes-Plattform wurde vollständig implementiert und evaluiert. Außerdem wurden mehrere Module realisiert. Das nachfolgende Beispielszenario demonstriert exemplarisch die Funktionsweise der Pipes-Plattform. Ein Großteil der hier verwendeten Module wurde in der Praxis bereits erfolgreich umgesetzt. In einem Raum sei ein DPWS-fähiges Ubisense-System, zwei WODA-Webcams und ein Bildschirm zum Anzeigen von Bildern installiert. Die Webcams sind so justiert, dass sie jeweils eine Hälfte des Raumes erfassen. Ein Benutzer, der mit einem Ubisense-Tag ausgestattet ist, bewegt sich frei im Raum. Ziel der Pipe soll es sein, immer das Bild derjenigen Webcam anzuzeigen, in deren Bereich sich der Benutzer gerade befindet.

Abbildung 7.8 zeigt einen möglichen Entwurf für diese Anwendung. Dieser besteht aus drei

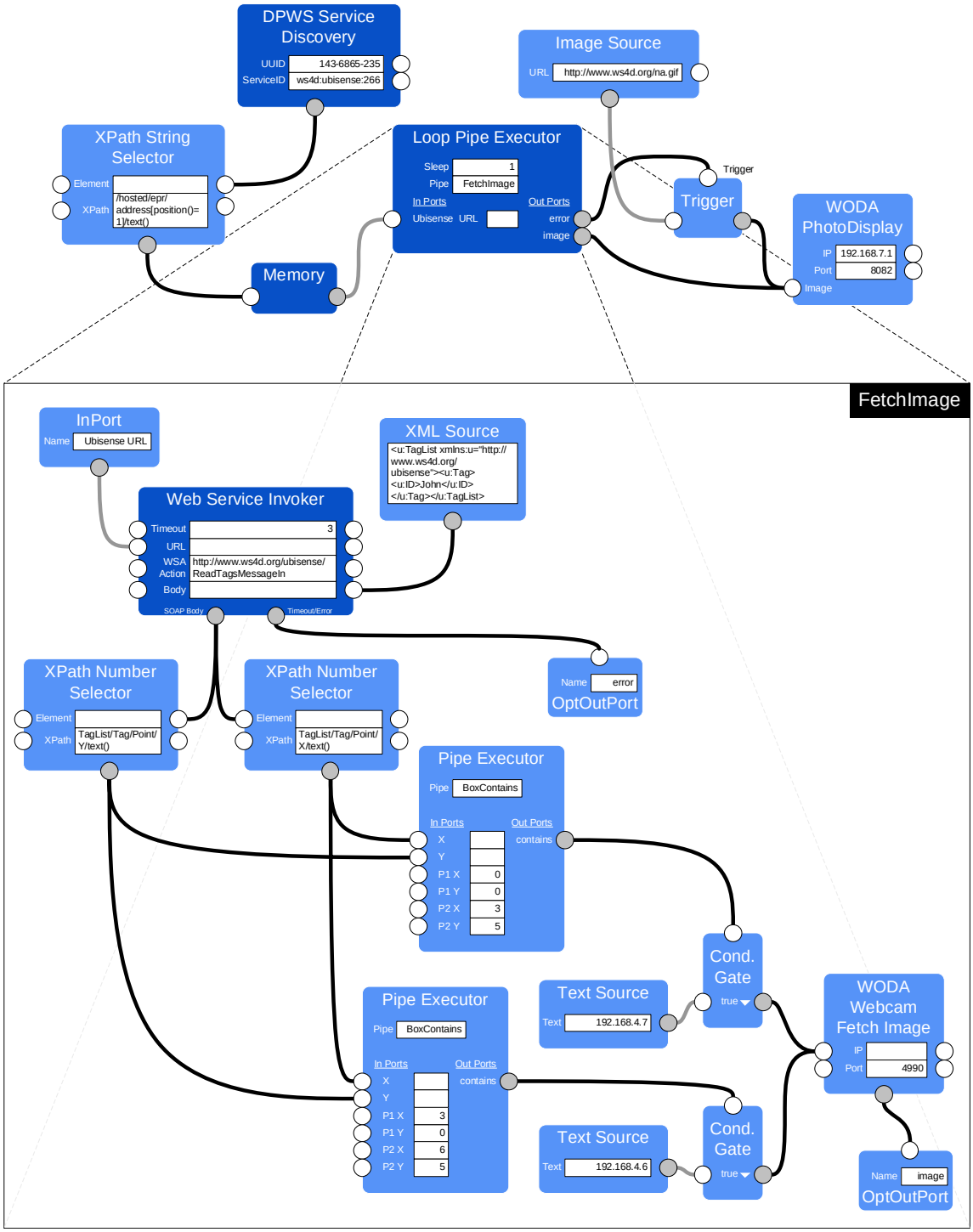

Abbildung 7.8: Pipes Beispielszenario

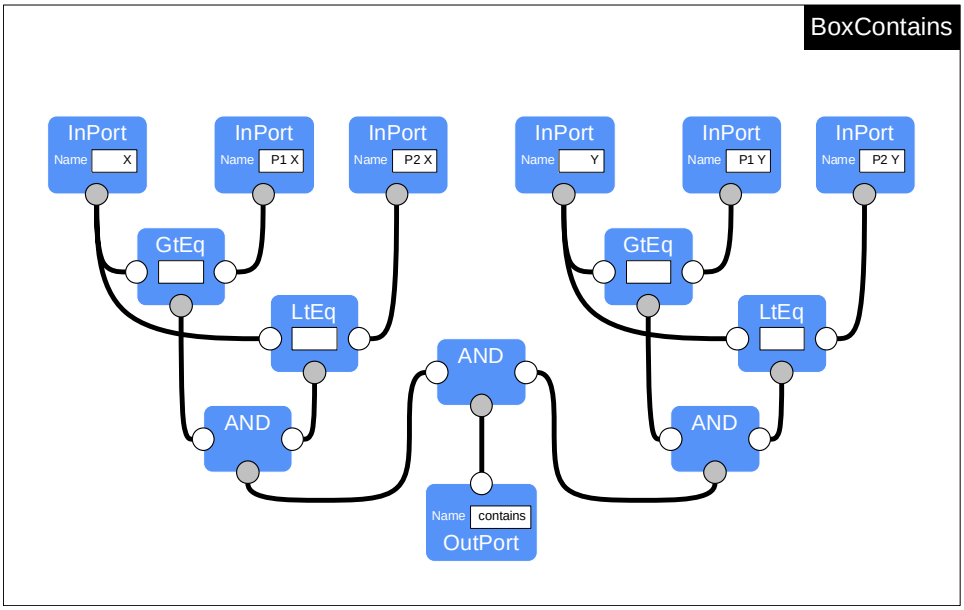

Abbildung 7.9: Pipes Beispielszenario (Fortsetzung)

Pipes, einer Hauptpipe und zwei Subpipes (*FetchImage* und *BoxContains*). Die Module der beiden Subpipes setzen bis auf *Web Services Invoker* alle das Default-Verhalten (Abs. 7.3.5) um.

In *FetchImage* sendet daher die *XML-Source* als erstes das statische XML-Dokument an den Web-Services-Invoker. Gleichzeitig fragt dieser über seinen aktiven *URL*-In-Port die URL des Ubisense-Dienstes an. Das XML-Dokument enthält den SOAP-Body für die *ReadTags*-Operation (Abs. 5.4.5). Die Tag-ID ist hier fest für den Benutzer *John* vorgegeben. Der *Web Services Invoker* beginnt seine Arbeit, wenn er die URL erhalten hat und ruft den Ubisense-Web-Service auf. Der Timeout ist auf 3 s eingestellt. Tritt der Timeout oder ein anderer Fehler auf, wird ein Fehlerdokument über den *Error*-Port gesendet. Im Erfolgsfall geht der SOAP-Body der Antwort, welcher die beiden aktuellen Koordinaten des Tags enthält, an die beiden *XPath-Number-Selector*-Module. Die Koordinaten (X und Y) werden extrahiert und an die *Pipes-Executor*-Module weitergereicht, die intern die Subpipe *BoxContains* aufrufen. *BoxContains* überprüft, in welchem der beiden Teilflächen sich das Tag derzeit befindet. Demzufolge wird einer der *contains*-Out-Ports *true*, der andere *false* sein. Die *Conditional-Gate*-Module bearbeiten keine Daten, sondern leiten sie lediglich weiter. Dazu muss am In-Port die Bedingung anliegen, für die das Modul konfiguriert wurde (im Beispiel lautet die Konfiguration *true*). Die beiden URLs der Webcams sind in den *Text-Source*-Modulen fest eingestellt. Einer der beiden URLs wird

angefragt (je nachdem welcher der *Conditional-Gates* schaltet) und an das *WODA-Webcam-Fetch-Image*-Modul gesendet. Dieses Modul kapselt die Webcam vollständig. Das eigentliche Kommunikationsprotokoll ist in diesem Fall auf Pipes-Ebene nicht sichtbar. Das aktuelle Bild wird schließlich an das *OptOutPort*-Modul gesendet.

Mit *InPort*- und *OutPort*/*OptOutPort*-Modulen lassen sich Ports definieren, die von Subpipes nach außen hin sichtbar sein sollen. Ein Modul, welches die Subpipe verwendet, braucht dann im Modeler auch nur diese entsprechenden Ports zu rendern. Der *Pipes-Executor* in *Fetch-Image* und der *Loop-Pipes-Executor* in der Hauptpipe sind Beispiele für solche Module. Die Implementierungen der *InPort*- und *OutPort*-Module reichen die Daten einfach nur durch. Die Unterscheidung zwischen *OutPort* und *OptOutPort* wurde hier vorgenommen, um verschiedene Ausführungsmodelle für eine Subpipe zu erlauben. Ein Modul, welches eine Subpipe verwendet, muss wissen, wann diese *fertig* ist, damit es sie beenden kann. Deklariert eine Subpipe beispielsweise nur *OutPort*-Module, implementiert sie damit das Default-Verhalten (Abs. 7.3.5).

Die *BoxContains*-Pipe würde in der Realität durch ein Modul implementiert werden, eine eigene Pipe ist hier nur zu Verständniszwecken modelliert worden. Die Pipe deklariert sechs In-Ports für die Koordinaten sowie ein Out-Port für das boolesche Ergebnis. Die Module *GtEq* (größer gleich), *LtEq* (kleiner gleich) und *AND* (logisches UND) sind selbsterklärend.

In der Hauptpipe überwacht das *DPWS-Service-Discovery*-Modul das Netzwerk auf den Ubisense-Dienst und schickt über seinen Out-Port *Hello*- und *Bye*-ähnliche Nachrichten. Der *XPath-String-Selector* extrahiert die URL aus der *Hello*-Nachricht, an der der Ubisense-Dienst zu erreichen ist. Für *Bye* wird ein leerer String zurückgegeben. Der *Memory* speichert immer genau ein Datenobjekt (das letzte), welches über seinen Out-Port angefragt werden kann.

Der *Loop-Pipe-Executor* fragt zunächst den *Memory* nach der URL ab, startet dann die *FetchImage*-Subpipe und beendet diese, wenn entweder der *error*- oder aber der *image*-Out-Port Daten sendet (beide Ports wurden in der Subpipe als *OptOutPort* deklariert). Anschließend wartet er 1 s (*Sleep*) und beginnt die Arbeit von vorne. Der *Trigger* funktioniert ähnlich wie das *Conditional-Gate*, jedoch schaltet er unabhängig von den Daten, die an seinem Trigger-Port liegen. War *FetchImage* erfolgreich, wird das Bild der Webcam, im Fehlerfall ein voreingestelltes Bild an das *WODA-PhotoDisplay* gesendet.

7.5 Zusammenfassung

Die *Pipes-Plattform* zielt auf eine wichtige Lücke im Umgang mit zukünftigen ubiquitären Geräten und Diensten ab: Für die Dienstnutzung werden ebenso Konzepte benötigt, wie für den eigentlichen Dienstentwurf. Diese Konzepte müssen die Charakteristiken, die ubiquitäres Com-

puting mit sich bringt, geeignet abdecken. Das sind vor allem Mobilität, Dynamik, heterogene Frameworks und die Benutzbarkeit durch weniger technisch versierte Benutzer.

Mit der Pipes-Plattform wurde ein eigener Ansatz vorgestellt, mit welchem sich neue Anwendungen, Prozesse oder Dienste grafisch modellieren lassen. Der Vorteil des Ansatzes besteht in der frameworkübergreifenden Verwendung von Diensten bzw. im domänenübergreifenden Einsatz durch die Verwendung der OSGi-Plattform. Weiterhin kann das Modellierungswerkzeug in einem herkömmlichen Browser verwendet werden. Dadurch ist eine hohe Akzeptanz bzw. Erreichbarkeit gewährleistet.

Es wurde die offene Plug-in-Schnittstelle beschrieben, über die zusätzliche, durch Dritte entwickelte Module der Plattform zur Verfügung gestellt werden können. Die Plattform unterstützt dadurch eine teamgerechte und offene Entwicklung, welche in den letzten Jahren immer wichtiger für verteilte Softwareentwicklung geworden ist. Beispielsweise kann dadurch die Zeit von der Markteinführung eines bestimmten Gerätes oder Dienstes bis zu dem Zeitpunkt, an dem dazu entsprechende Module zur Verfügung gestellt werden können, erheblich reduziert werden. Dieses Prinzip wirkt gleichzeitig als ein zuverlässiger Filter, da sich nur geeignete Module durchsetzen werden.

Ein abschließendes Beispiel demonstrierte, wie sich mittels Pipes eine völlig neue Anwendung erstellen lässt, die Daten, Dienste und Geräte verwendet, ohne eine Zeile Code schreiben zu müssen.

8 Zusammenfassung und Ausblick

8.1 Zusammenfassung

Mit der Allgegenwärtigkeit eingebetteter Systeme steigt der Bedarf, diese in unterschiedlichen Kontexten und komplexen Szenarien zu nutzen. Serviceorientierte Architekturen (SOA) gelten hier als ein aussichtsreiches Paradigma, um bestehende und neue Anwendungen, Dienste und Geräte über administrative und technische Grenzen hinaus zu integrieren und zu nutzen und dabei gleichzeitig die Softwarekomplexität zu senken.

Um eine Vielzahl von Geräten und Diensten unterschiedlichster Anwendungsgebiete zu integrieren, bedarf es universell einsetzbarer bzw. domänenübergreifender Frameworks. Hardware-, Plattform- und Programmiersprachenunabhängigkeit sind unentbehrliche Anforderungen an derartige Frameworks. Um eine hohe Akzeptanz zu gewährleisten, sollten sie ferner auf offenen, erweiterbaren Spezifikationen beruhen, internetfähig sein und die dynamische Entdeckung (Discovery) fördern.

Der Schwerpunkt der Dissertation liegt auf Konzeption, Umsetzung sowie Anwendung von Frameworks, die Charakteristiken serviceorientierter Architekturen (SOA) aufweisen und Geräte integrieren, dem SOA-basierten Aufbau einer Lokalisierungsplattform und der grafischen Anwendungs- bzw. Prozessmodellierung unter Beachtung einer jeweils möglichst universellen Einsetzbarkeit. Durch diese vertikale Vorgehensweise wurden exemplarisch Probleme und Lösungen für diese drei horizontalen Ebenen der Frameworks/Paradigmen, Dienste sowie Anwendungsdesign vorgestellt.

Es wurden zunächst Anforderungen beschrieben, die universell einsetzbare Frameworks besitzen sollten. Zu den wichtigsten zählen die Unabhängigkeit von spezifischer Hardware, Plattform und Programmiersprache, die automatische Entdeckung von Geräten und Diensten (Discovery), die Überwachung von Ereignissen, Internetfähigkeit sowie eine breite Unterstützung durch Werkzeuge (z. B. Codegeneratoren, Webbrowser usw.).

Viele der existierenden Lösungen, wie z. B. Jini, OSGi oder UPnP, weisen jedoch Abhängigkeiten zu Programmiersprachen (OSGi, EJB, Jini) oder zu Programmierkonzepten (RPC/RMI in UPnP und Jini, Objektorientierung in CORBA) auf, oder aber sie adressieren nur ausgewählte Domänen wie die Unterhaltungselektronik (HAVi), die Gebäudeautomatisierung (EIB) oder die Steuerung von Industrieanlagen (CAN, PROFIBUS). Das Devices Profile for Web Ser-

vices (DPWS) und Universal Plug and Play (UPnP) sind zwei Standards, die viele der genannten Anforderungen unterstützen und auch für eine domänenübergreifende Anwendbarkeit geeignet sind.

Um DPWS praktisch evaluieren zu können, wurde die eigene, auf Java basierende DPWS-Implementierung für das Apache-Axis2-Projekt vorgestellt, welche innerhalb der WS4D-Initiative entwickelt wurde. Die dafür getroffenen Designentscheidungen wurden begründet und in der Praxis evaluiert. Die Implementierung setzt dazu die einzelnen Web-Services-Spezifikationen, die für DPWS benötigt werden, modular um, so dass diese zukünftig auch von anderen Web-Services-Profilen oder anderen Anwendungen benutzt werden können. Damit wurde der Web-Services-Ansatz, Profile bzw. Anwendungen aus miteinander kombinierbaren Protokollen bzw. Softwaremodulen zu erstellen, auch bei der Profilimplementierung konsequent fortgesetzt. Mit der DPWS-Umsetzung für Axis2 ist es nun erstmals möglich DPWS-Geräte in Enterprise-Systemen, in denen Axis2 eingesetzt wird, zu verwenden.

Beim Aufbau einer Lokalisierungsplattform wurde gezeigt, wie geeigneter Dienstentwurf und Ausnutzung des SOA-Paradigmas beim Aufbau einer auf Langfristigkeit und Wiederverwendbarkeit ausgelegten IT-Infrastruktur helfen kann. Unter der Verwendung des Devices Profile for Web Services wurden Möglichkeiten erörtert, die SOA auszubauen und gleichzeitig intakt zu lassen, so dass Anwendungen und Dienste jederzeit weiter funktionieren konnten. Die einzelnen Schritte wurden durch eine zuvor aufgestellte Klassifizierung von SOA-Teilnehmern visuell als SOA-Diagramme aufbereitet.

Außerdem kam der eigene Typisierungsmechanismus zum Einsatz, mit welchem Geräte- und Diensttypen für DPWS formal definiert werden können. Ein entsprechender Mechanismus fehlte bisher. Durch die Verwendung der XML-Notation kann aus den Beschreibungsdokumenten Code generiert werden, wodurch die Entwicklung von Diensten und Dienstnutzern vereinfacht wird. Der Ansatz unterstützt weiterhin die Versionierung von Typen, indem diese kompatibel zu anderen gekennzeichnet werden können. Dadurch ist gewährleistet, dass Dienstnutzer, die für ein älteres Anwendungsprotokoll implementiert wurden, auch weiterhin abwärtskompatible Geräte und Dienste finden können.

Schließlich wurden Details der prototypischen Anbindung der Lokalisierungssysteme BlueScan und Ubisense vorgestellt, die in der Praxis erfolgreich getestet werden konnten. Dabei konnten durch BlueScan alle zuvor identifizierten Nachteile des BlueTrack-Systems behoben werden. Weiterhin wurde ein Ansatz für den Schnittstellenentwurf eines Lokalisierungsdienstes vorgestellt, welcher heterogene Lokalisierungssysteme integriert und nach außen homogen abbildet und dadurch auch zukünftig mit neuen Systemen erweitert werden kann. Einige Details aus der praktischen Umsetzung der BlueScan-Geräte, des BlueScan-Servers, der Anbindung des Ubisen-

se-Systems, des Discovery-Proxies und des Location-Services wurden schließlich beschrieben.

Von der Erkenntnis ausgehend, dass das Internet mit seinen etablierten Standards wie URI, DNS und HTTP selbst eine global verfügbare und universelle Plattform darstellt, wurde mit der Web-oriented Device Architecture (WODA) konzeptionell ein eigener, alternativer Ansatz für ein Integrationsframework präsentiert. Für das REST-konforme Framework WODA wurden dazu Lösungen entwickelt, die die Grundprinzipien einer serviceorientierten Gerätearchitektur umsetzen.

REST (Representational State Transfer) liefert dazu die architektonische Grundlage. Für WODA bedeutet dieses, dass alle Komponenten wie Geräte, Dienste, Diensttypen, Abonnements für Ereignisse usw. als Ressourcen modelliert werden und auf diese mit dem HTTP-Protokoll semantisch fest definierte Methoden aufgerufen werden können. Der Vorteil dieses Ansatzes ist die Gleichbehandlung aller Ressourcen auf Anwendungsebene, wodurch gleiche Werkzeuge (z. B. Firewalls, Caches, Webbrowser usw.) unabhängig von konkreten Diensten oder Ressourcen anwendbar sind.

Für Addressing und Discovery setzt WODA mit Zeroconf auf das internetweit etablierte DNS-System und bietet damit besondere Vorteile im Vergleich zu den Konkurrenztechnologien DPWS und UPnP. Zeroconf ist bereits in vielen Produkten integriert und in der Praxis erprobt. Es sind außerdem einige, u. a. freie, Implementierungen verfügbar. Es wurde ein spezieller DNS-Servicetyp für WODA als Subtyp von HTTP deklariert. Mittels Zeroconf werden daher bei einer WODA-Suche bereits WODA- bzw. HTTP-Ressourcen gefunden.

Für die Dienstbeschreibung wurde die *Web Application Description Language* (WADL) verwendet. Sie wurde ausgewählt, da sie viele Möglichkeiten zur Beschreibung von Ressourcen anbietet, in XML notiert ist und da bereits ein Codegenerator (für Clients) existiert. Sie ist außerdem erweiterbar hinsichtlich der HTTP-Methoden. Dieses war eine wichtige Anforderung, um Ressourcen beschreiben zu können, die den WODA-Ereignismechanismus verwenden, da dieser eine neue HTTP-Methode definiert. Für die Gerätebeschreibung wurde ein eigenes XML-Schema definiert und es wurde beschrieben wie die Ressource, die die Gerätebeschreibung enthält, ermittelt werden kann.

Mit dem HTTP-basierten Publish-Subscribe-Mechanismus wurde auf Basisebene eine einfache Möglichkeit beschrieben, Ereignisse zu abonnieren und zu veröffentlichen. Auch hier wurde das REST-Konzept umgesetzt, indem Abonnements als Ressourcen abgebildet werden und dadurch mit den gleichen Mitteln und Werkzeugen bearbeitet werden können. Es wurde gezeigt wie anwendungsspezifische Erweiterungen (z. B. Filter- und Lease-Mechanismen) REST-konform umgesetzt werden können. Im Gegensatz zu UPnP erlaubt die vorgestellte Lösung die Implementierung der beiden Ereignismodelle Peer-to-Peer und Vermittler. Ein weiterer Vorteil

besteht in der Möglichkeit die zu überwachenden Ressourcen sowohl vom Dienst als auch vom Dienstnutzer definieren zu lassen. Für letzteres wurde mit der *Resource Factory* ein einfaches Pattern vorgestellt.

Durch den Einsatz der Ajax-Technologie in der Presentation-Phase wird eine ressourcenschonende Nutzung von Geräten durch herkömmliche Webbrowser ermöglicht. WODA benötigt im Gegensatz zu UPnP und DPWS keine zusätzliche Schnittstelle für die Präsentation, sondern kann die Anwendungsressourcen der Dienstimplementierung wiederverwenden. WODA realisiert damit aus SOA-Sicht einen verbesserten Ansatz, da Dienste nur *eine* Schnittstelle unabhängig von der Art des Dienstnutzers anbieten müssen.

In WODA existieren Geräte nur noch als Gerätebeschreibung. Im Gegensatz zu UPnP und DPWS wird ein Gerät ausschließlich über seine Dienstinstanzen identifiziert. Eine weitere Unterscheidung ist nicht notwendig, da Dienstnutzer entweder Dienste eines bestimmten Typs oder aber eine bestimmte Dienstinstanz suchen und benutzen. In WODA ist eine Geräteinstanz gleichbedeutend mit einer Dienstinstanz, die auf dem Gerät läuft.

Für die grafische Modellierung von Anwendungen bzw. Prozessen, die Geräte und Dienste verwenden oder neue Dienste erzeugen, wurde die selbst entwickelte Pipes-Plattform vorgestellt. Der Vorteil des Lösung besteht in der frameworkübergreifenden Verwendung von Geräten und Diensten und im domänenübergreifenden Einsatz durch die Verwendung der OSGi-Plattform.

Die Pipes-Plattform verwendet das Konzept der autonomen Komponenten (Module), die Ein- und Ausgangsports besitzen und über Kabel miteinander verbunden werden können. Über diese können Module Daten senden und abfragen. Die dabei entstehenden Pipes stellen Anwendungen oder Prozesse dar und können auf der Pipes-Plattform zentral ausgeführt werden. Der Ansatz der autonomen Komponenten erlaubt den Modulen unabhängig von anderen Modulen Daten zu senden und abzufragen. Um synchronisierte Abläufe realisieren zu können, wurde daher mit dem Default Execution Model ein fest definiertes Verhalten vorgestellt und implementiert, welches von Modulen verwendet werden kann. Module kapseln technologie- bzw. domänenspezifische Bibliotheken und werden als OSGi-Bundles installiert.

Die Pipes-Plattform trennt durchgehend zwischen technisch versierten Bibliotheks- und Modul-Entwicklern, die Bibliotheken und Module bereitstellen, sowie technisch weniger versierten Pipes-Entwicklern, die Module in einer grafischen Oberfläche verwenden. Das Modellierungswerkzeug kann in einem herkömmlichen Webbrowser benutzt werden. Dadurch ist eine hohe Akzeptanz bzw. Erreichbarkeit gewährleistet.

Weiterhin wurde die offene Plug-in-Schnittstelle beschrieben, über die zusätzliche, durch Drittanbieter entwickelte Module der Plattform zur Verfügung gestellt werden können. Die Plattform unterstützt dadurch eine teamgerechte und offene Entwicklung, welche in den letzten Jahren

immer wichtiger für Softwareentwicklung geworden ist.

Die Pipes-Plattform wurde vollständig implementiert und mit mehreren Modulen evaluiert. Sie schließt eine wichtige Lücke im Umgang mit der benutzerspezifischen Verwendung von Diensten und Geräten. Dazu adressiert sie explizit technisch weniger versierte Benutzer, die auch ohne Programmierkenntnisse und ohne sich mit unterschiedlichen Frameworks auskennen zu müssen Anwendungen erstellen können.

8.2 Ausblick

Das Devices Profile for Web Services ist ein wichtiges Profil innerhalb der Web Services Spezifikationen. Es wird in der Zukunft jedoch noch mehreren Änderungen unterliegen, da es in der aktuellen Fassung teilweise auf ungültige Spezifikationen basiert. Die Entwicklung der Web Services schritt langsamer voran als ursprünglich beabsichtigt, es sind jedoch erste Konsolidierungen zu verzeichnen. Zusammen mit der breiten Unterstützung aus der Industrie und Wirtschaft besitzt DPWS daher langfristig betrachtet ein sehr großes Potenzial.

Das Internet stellt mit seinen etablierten Protokollen eine global verfügbare Plattform dar. Für diese lässt sich, wie mit WODA gezeigt, ebenfalls ein konkurrenzfähiges, alternatives Integrationsframework entwerfen, indem geeignete Mechanismen gefunden werden, die die Vorteile der bestehenden Infrastruktur nutzen anstatt sie zu umgehen.

Literaturverzeichnis

[1] Der Spiegel. Elektronenroboter in Deutschland. `http://www.tecchannel.de/server/hardware/466465/index4.html`, Mai 1965.

[2] TecChannel. Die zehn größten IT-Irrtümer und -Fehlprognosen. `http://www.tecchannel.de/server/hardware/466465/`, April 2007.

[3] Leo A. Nefiodow. *Der sechste Kondratieff*. Rhein-Sieg Verlag, 2001.

[4] stern.de. Interview mit Gordon Moore: "Mir ging es um etwas ganz anderes". `http://www.stern.de/computer-technik/computer/:Gordon-Moore-Mir/539237.html`, April 2005.

[5] Brett A. Warneke and Kristofer S.J. Pister. An Ultra-Low Energy Microcontroller for Smart Dust Wireless Sensor Networks. In *IEEE International Solid-State Circuits Conference*, Februar 2004.

[6] FOX Board, Acme Systems srl. URL `http://www.acmesystems.it/`.

[7] Steve Mann. Wearable Computing as means for Personal Empowerment. `http://wearcam.org/icwckeynote.html`, Mai 1998. Keynote Address for The First International Conference on Wearable Computing, ICWC-98, May 12-13, Fairfax VA.

[8] Mark Weiser. The computer for the 21st century. *Human-computer interaction: toward the year 2000*, pages 933–940, 1995.

[9] Henning Schulzrinne. The Vision and Reality of Ubiquitous Computing (Keynote slides). `http://www.cs.columbia.edu/~hgs/papers/2007/mobiquitous-keynote.ppt`, August 2007. The 4th Annual International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MOBIQUITOUS 2007), Philadelphia, PA.

[10] Nokia Sensor – Se and be seen. `http://www.nokia-asia.com/A4416020`, 2005.

[11] Staffan Björk, Jussi Holopainen, Peter Ljungstrand, and Regan Mandryk. Special issue on ubiquitous games. *Personal Ubiquitous Comput.*, 6(5-6):358–361, 2002.

[12] Marc Haase, Matthias Handy, and Dirk Timmermann. Bluetrack – Imperceptible Tracking of Bluetooth Devices. In *UbiComp 2004 Adjunct Proceedings*, Nottingham, Great Britain, UK, September 2004. The Sixth International Conference on Ubiquitous Computing.

[13] Bluetooth SIG. *Specification of the Bluetooth System 1.2*, 2003.

[14] BlueZ – Official Linux Bluetooth protocol stack. `http://www.bluez.org/`.

[15] Christian Lüders. *Lokale Funknetze – Wireless LANs (IEE 802.11), Bluetooth, DECT.* Vogel Industrie Medien, 2007.

[16] LOMS – Local Mobile Services, 2007. `http://www.loms-itea.org`.

[17] Tim Berners-Lee. *Weaving the Web – The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor.* Harper San Francisco, 1999.

[18] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. *Uniform Resource Identifier (URI): Generic Syntax.* Network Working Group, Request for Comments: 3986, Januar 2005. `http://tools.ietf.org/html/rfc3986`.

[19] Roy T. Fielding, Jim Gettys, Jeff Mogul, Henry Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1.* Network Working Group, Request for Comments: 2616, Juni 1999. `http://www.ietf.org/rfc/rfc2616.txt`.

[20] Ryan Asleson and Nathaniel T. Schutta. *Foundations of Ajax.* Apress, Oktober 2005.

[21] JavaScript Object Notation. `http://www.json.org/`.

[22] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design.* Prentice Hall PTR, August 2005.

[23] Mahesh Dodani. From Objects to Services: A Journey in Search of Component Reuse Nirvana. *Journal of Object Technology*, 3(8):49–54, September-Oktober 2004. `http://www.jot.fm/issues/issue_2004_09/column5`.

[24] Sayed Hashimi. Service-oriented Architecture Explained, Mai 2004. `http://dev2dev.bea.com/pub/a/2004/05/soa_hashimi.html`.

[25] Wolfgang Dostal, Mario Jeckle, and Ingo Melzer. *Service-orientierte Architekturen mit Web Services. Konzepte - Standards - Praxis.* Spektrum Akademischer Verlag, September 2005.

[26] James McGoverna, Sameer Tyagi, Michael Stevens, and Sunil Mathew. *Java Web Services Architecture*. Morgan Kaufmann, Juli 2003. `http://java.sun.com/developer/Books/j2ee/jwsa/index.html`.

[27] Dirk Krafzig, Karl Banke, and Dirk Slama. *Enterprise SOA: Service-Oriented Architecture Best Practices*. Prentice Hall PTR, November 2004.

[28] Helmut Balzert. *Lehrbuch der Software-Technik (Software-Entwicklung)*. Spektrum Akademischer Verlag, 2000.

[29] Grady Booch. *Object-oriented Analysis and Design with Applications*. Benjamin-Cummings Publishing, 1993.

[30] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional, Boston, 2002.

[31] Andrew Tanenbaum; Marten van Steen. *Verteilte Systeme, Grundlagen und Paradigmen*. Prentice Hall, 2003.

[32] CORBA FAQ. `http://www.omg.org/gettingstarted/corbafaq.htm`.

[33] Remote method invocation home. `http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp`.

[34] Enterprise javabeans technology. `http://java.sun.com/products/ejb/`.

[35] Com: Component object model technologies. `http://www.microsoft.com/com/default.mspx`.

[36] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Donald F. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall PTR, 2005.

[37] Marco Taisch and Armando Colombo. The Socrades European project (service-orientated cross-layer infRAstructure for distributed smart embedded devices). `http://www.socrades.eu/Documents/objects/file1201101964.73`, Juni 2007. Second World Congress on Engineering Asset Management and the Fourth International Conference on Condition Monitoring, WCEAM CM 2007, Harrogate, UK.

[38] Web Services Standards as of Q1 2007. `http://www.innoq.com/soa/ws-standards/poster/`.

[39] Organization for the Advancement of Structured Information Standards (OASIS). URL `http://www.oasis-open.org/home/index.php`.

[40] Web Services Interoperability Organization (WS-I). URL `http://www.ws-i.org/`.

[41] World Wide Web Consortium (W3C). URL `http://www.w3.org/`.

[42] Internet Engineering Task Force (IETF). URL `http://www.ietf.org/`.

[43] W3C – World Wide Web Consortium. *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*, April 2007. `http://www.w3.org/TR/2007/REC-soap12-part1-20070427`.

[44] Mark Baker and Mark Nottingham. *The "application/soap+xml" media type.* Network Working Group, Request for Comments: 3902, September 2004. `http://www.ietf.org/rfc/rfc3902.txt`.

[45] *SOAP-over-UDP*, September 2004. `http://schemas.xmlsoap.org/ws/2004/09/soap-over-udp/`.

[46] W3C – World Wide Web Consortium. *Web Services Description Language (WSDL) 1.1*, März 2001. `http://www.w3.org/TR/2001/NOTE-wsdl-20010315`.

[47] Russell Butek. Which style of WSDL should I use? URL `http://www-128.ibm.com/developerworks/webservices/library/ws-whichwsdl/`, 2005.

[48] Online Community for the Universal Description, Discovery and Integration. `http://uddi.xml.org/`.

[49] Madeleine Wright. A Detailed Investigation of Interoperability for Web Services. Dissertation, Rhodes University, Dezember 2005.

[50] W3C Submission. *Web Services Addressing (WS-Addressing)*, August 2004. `http://www.w3.org/Submission/ws-addressing/`.

[51] *Web Services Dynamic Discovery (WS-Discovery)*, April 2005. `http://schemas.xmlsoap.org/ws/2005/04/discovery/`.

[52] *Web Services Eventing (WS-Eventing)*, August 2004. `http://schemas.xmlsoap.org/ws/2004/08/eventing/`.

[53] Roy Thomas Fielding. Architectural styles and the design of network-based software architectures. Dissertation, University of California, Irvine, 2000.

[54] Stuart Cheshire and Daniel H. Steinberg. *Zero Configuration Networking: The Definitive Guide*. O'Reilly, Dezember 2005.

[55] R. Droms. *Dynamic Host Configuration Protocol*. Network Working Group, Request for Comments: 2131, 1997. `http://www.faqs.org/rfcs/rfc2131.html`.

[56] S. Cheshire, B. Aboba, and E. Guttman. *Dynamic Configuration of IPv4 Link-Local Addresses*. Network Working Group, Request for Comments: 3927, 2005. `http://www.faqs.org/rfcs/rfc3927.html`.

[57] Stuart Cheshire and Marc Krochmal. *Multicast DNS*. Internet Draft, August 2006. `http://files.multicastdns.org/draft-cheshire-dnsext-multicastdns.txt`.

[58] Marc J. Hadley. *Web Application Description Language (WADL)*. Sun Microsystems Inc., November 2006. `https://wadl.dev.java.net/wadl20061109.pdf`.

[59] Web Application Description Language (WADL) – Specifications and Tools. URL `https://wadl.dev.java.net/`.

[60] Universal Plug and Play Forum. URL `http://www.upnp.org`.

[61] UPnP Forum. *UPnP Device Architecture v.1.0.1*, 2006. `http://www.upnp.org/specs/arch/UPnP-DeviceArchitecture-v1.0-20060720.pdf`.

[62] Yaron Y. Goland, Ting Cai, Paul Leach, Ye Gu, and Shivaun Albright. *Simple Service Discovery Protocol/1.0*. Internet Draft, 2000. `http://quimby.gnus.org/internet-drafts/draft-cai-ssdp-v1-03.txt`.

[63] J. Cohen, S. Aggarwal, and Y. Y. Goland. *General Event Notification Architecture*. Internet Draft, 2000. `http://quimby.gnus.org/internet-drafts/draft-cohen-gena-p-base-01.txt`, `http://quimby.gnus.org/internet-drafts/draft-cohen-gena-client-00.txt`.

[64] *Devices Profile for Web Services*, Februar 2006. `http://schemas.xmlsoap.org/ws/2006/02/devprof/`.

[65] *Web Services Metadata Exchange (WS-MetadataExchange)*, September 2004. `http://schemas.xmlsoap.org/ws/2004/09/mex/`.

[66] *Web Services Transfer (WS-Transfer)*, September 2004. `http://schemas.xmlsoap.org/ws/2004/09/transfer/`.

[67] LON. URL `http://www.echelon.com/default.htm`.

[68] Michael Rose, Werner Kriesel, and Jens Rennefahrt. *EIB für die Gebäudesystemtechnik in Wohn- und Zweckbau.* Hüthig, Juni 2000.

[69] PROFIBUS. URL `http://www.profibus.com/`.

[70] Andreas Lorinser, Christian Schlegel, Tom A. H. M. Suters, and Konrad Etschberger. *CAN Controller Area Network. Grundlagen, Protokolle, Bausteine, Anwendungen.* Hanser Fachbuch, 1994.

[71] Grzemba and von der Wense. *LIN-Bus.* Franzis, Januar 2005.

[72] FlexRay – The communication system for advanced automotive control applications. URL `http://www.flexray.com/`.

[73] Mathias Rausch. *FlexRay. Grundlagen, Funktionsweise, Anwendung.* Hanser Fachbuch, November 2007.

[74] Bob Pascoe. Salutation Architectures and the newly defined service discovery protocols from Microsoft and Sun – How does the Salutation Architecture stack up – A Salutation White Paper, Juni 1999. `http://web.archive.org/web/20050227213123/www.salutation.org/whitepaper/Jini-UPnP.PDF`.

[75] Olivier Dubuisson. *ASN.1 - Communication Between Heterogeneous Systems.* Morgan Kaufmann Publishers, September 2000. `http://www.oss.com/asn1/dubuisson.html`.

[76] J. Teirikangas. HAVi: Home Audio Video Interoperability, 2001. Technical report.

[77] Simon Gibbs, Ravi Gauba, Ram Balaraman, and Rodger Lea. *HAVi Example By Example: Java Programming for Home Entertainment Devices.* Prentice Hall, August 2001.

[78] Sun Microsystems. *Jini Architecture Specification Version 1.2*, 2001.

[79] OSGi Alliance. URL `http://www.osgi.org/`.

[80] Dave Marples and Peter Kriens. The Open Services Gateway Initiative: An Introductory Overview. In *IEEE Communications Magazine*, volume 39, pages 110–114, Dezember 2001.

[81] Eclipse – Equinox. URL `http://www.eclipse.org/equinox/`.

[82] Object Management Group. *Common Object Request Broker Architecture: Core Specification, Version 3.0.3.* `http://www.omg.org/technology/documents/corba_spec_catalog.htm`.

[83] James Neil Weatherall. An Embedded Ubiquitous Control Architecture for Low Power Systems. Master's thesis, Queens' College, Juli 2002.

[84] Steffen Deter. Plug-and-Participate for Limited Devices in the Field of Industrial Automation. Dissertation, Fachbereich Mathematik und Informatik der Philipps-Universität Marburg, 2003.

[85] James Beck, Alain Gefflaut, and Nayeem Islam. MOCA: A Service Framework for Mobile Computing Devices. In *MobiDe '99: Proceedings of the 1st ACM international workshop on Data engineering for wireless and mobile access*, pages 62–68, New York, NY, USA, 1999. ACM.

[86] SIRENA: Service Infrastructure for Real-time Embedded Networked Applications. `http://www.sirena-itea.org`, 2006.

[87] Hendrik Bohn, Andreas Bobek, and Frank Golatowski. SIRENA – Service Infrastructure for Real-time Embedded Networked Devices: A service oriented framework for different domains. In *5th International Conference on Networking (ICN 06)*, April 2006.

[88] Andreas Bobek, Hendrik Bohn, and Frank Golatowski. UPnP AV Architecture – Generic Interface Design and Java Implementation. In *23rd IASTED International Multi-Conference on Applied Informatics*, pages 699–704, Februar 2005.

[89] Michael Ditze, Chris Loeser, Hendrik Bohn, Andreas Bobek, and Frank Golatowski. Quality of Service and Proactive Content Replication in UPnP based A/V Environments. In *23rd IASTED International Multi-Conference on Applied Informatics*, pages 729–734, Februar 2005.

[90] Andreas Bobek, Hendrik Bohn, Frank Golatowski, Gerd Kachel, and Andreas Spreen. Enabling Workflow in UPnP Networks. In *3rd IEEE International Conference on Industrial Informatics, INDIN 05*, August 2005.

[91] Stefan Illner, Heiko Krumm, Ingo Lück, Andre Pohl, Andreas Bobek, Hendrik Bohn, and Frank Golatowski. Management of Embedded Service Systems – An Applied Approach.

In *20th IEEE International Conference on Advanced Information Networking and Applications (AINA 06)*, pages 519–523, April 2006.

[92] Hendrik Bohn, Andreas Bobek, and Frank Golatowski. Bluetooth Device Manager Connecting a Large Number of Resource-Constraint Devices in a Service-Oriented Bluetooth Network. In *4th IEEE International Conference on Networking (ICN 05)*, pages 430–437, April 2005.

[93] WS4D – Web Services for Devices. `http://www.ws4d.org`, 2007.

[94] Elmar Zeeb, Andreas Bobek, Hendrik Bohn, and Frank Golatowski. Lessons learned from implementing the Devices Profile for Web Services. In *Inaugural IEEE International Conference on Digital Ecosystems and Technologies (IEEE DEST 2007)*, pages 229–232, Februar 2007.

[95] Elmar Zeeb, Andreas Bobek, Hendrik Bohn, and Frank Golatowski. Service-Oriented Architectures for Embedded Systems Using Devices Profile for Web Services. In *2nd International IEEE Workshop on Service Oriented Architectures in Converging Networked Environments (SOCNE 07)*, Mai 2007.

[96] Elmar Zeeb, Andreas Bobek, Frank Golatowski, and Dirk Timmermann. Web Services – zu groß für eingebettete Systeme? In *12. Maritimes Symposium Elektrotechnik, Elektronik und Informationstechnik*, pages 179–184, Oktober 2007.

[97] Elmar Zeeb, Andreas Bobek, Hendrik Bohn, Steffen Prüter, Andre Pohl, Heiko Krumm, Ingo Lück, Frank Golatowski, and Dirk Timmermann. WS4D: SOA-Toolkits making embedded systems ready for Web Services. In *Open Source Software and Productlines 2007 (OSSPL07)*, Juni 2007.

[98] M. Asif, S. Majumdar, and R. Dragnea. Hosting Web Services on Resource Constrained Devices. In *IEEE International Conference on Web Services 2007, ICWS 2007*, pages 583–590, Juli 2007.

[99] Witold Drytkiewicz, Ilja Radusch, Stefan Arbanowski, and Radu Popescu-Zeletin. pREST: A REST-based Protocol for Pervasive Systems. In *MASS*, pages 340–348, Oktober 2004.

[100] Jan Newmarch. A RESTful Approach: Clean UPnP without SOAP. In *Second IEEE Consumer Communications and Networking Conference, CCNC 2005*, pages 134–138, Januar 2005.

[101] Workshop on Internet Scale Event Notification (WISEN). URL `http://www.isr.uci.edu/events/twist/wisen98/`, Juli 1998.

[102] Lucas Gonze. A Secure Protocol for Desktop Web Servers. URL `http://www.gonze.com/http-notifications.html`, September 2003.

[103] Alessandro Alinone. Changing the Web Paradigm – Moving from traditional Web applications to Streaming-AJAX, 2006. `http://www.lightstreamer.com/Lightstreamer_Paradigm.pdf`.

[104] N.A.B. Gray. Comparison of Web Services, Java-RMI, and CORBA service implementations. In *Fifth Australasian Workshop on Software and System Architectures*, 2004.

[105] Dionissis Vassilopoulos, Thomi Pilioura, and Aphrodite Tsalgatidou. Distributed Technologies – CORBA, Enterprise JavaBeans, Web Services – A Comparative Presentation. In *PDP '06: Proceedings of the 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 280–284, Washington, DC, USA, 2006. IEEE Computer Society.

[106] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. RESTful Web Services vs. Big Web Services: Making the Right Architectural Decision. In *17th International World Wide Web Conference, WWW2008*, April 2008.

[107] Ubisense. URL `http://www.ubisense.de/`.

[108] Open Mobile Alliance. URL `http://www.openmobilealliance.org/default.aspx`.

[109] Axel Küpper, Georg Treu, and Claudia Linnhoff-Popien. TraX: a device-centric middleware framework for location-based services. In *IEEE Communications Magazine*, pages 114–120, September 2006.

[110] Anthony LaMarca, Yatin Chawathe, Sunny Consolvo, Jeffrey Hightower, Ian Smith, James Scott, Tim Sohn, James Howard, Jeff Hughes, Fred Potter, Jason Tabert, Pauline Powledge, Gaetano Borriello, and Bill Schilit. Place Lab: Device Positioning Using Radio Beacons in the Wild. In *International Conference on Pervasive Computing*, pages 116–133, Mai 2005.

[111] Place Lab – a privacy-observant location system. URL `http://www.placelab.org/`.

[112] A.J.H. Peddemors, M.M. Lankhorst, and J. de Heer. Presence, location and instant messaging in a context-aware application framework. In *4th International Conference on Mobile Data Management (MDM2003)*, Januar 2003.

[113] Johan de Heer, Ronald van Eijk, Petri Määttä, and Arjan Peddemors. Project GigaMobile – A generic interface for location handling, 2002. D4.3, Task 4.1.

[114] Anand Ranganathan, Jalal Al-Muhtadi, Shiva Chetan, Roy Campbell, and M. Dennis Mickunas. MiddleWhere: A Middleware for Location Awareness in Ubiquitous Computing Applications. In *ACM/IFIP/USENIX 5th International Middleware Conference*, number 78, pages 397–416, Oktober 2004.

[115] Frank Dürr, Nicola Hönle, Daniela Nicklas, Christian Becker, and Kurt Rothermel. Nexus– A Platform for Context-Aware Applications. In *1. Fachgespräch Ortsbezogene Anwendungen und Dienste der GI-Fachgruppe KuVS*, 2004.

[116] Matthias Wieland and Daniela Nicklas. Ein Framework für kontextbezogene Anwendungen in der Nexus-Plattform. In *3. GI/ITG KuVS Fachgespräch: Ortsbezogene Anwendungen und Dienste*, 2006.

[117] Alexander Leonhardi. Architektur eines verteilten skalierbaren Lokationsdienstes. Master's thesis, Fakultät Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart, Juni 2003.

[118] Ulf Leonhardt. Supporting Location-Awareness in Open Distributed Systems. Dissertation, Department of Computing Imperial College of Science, Technology and Medicine University of London, Mai 1998.

[119] Matjaz B. Juric, Benny Mathew, and Poornachandra Sarang. *Business Process Execution Language for Web Services 2nd Edition.* Packt Pub, Januar 2006.

[120] Oracle SOA Suite – Oracle BPEL Process Manager, Juli 2006.

[121] Bruce Silver. IBM WebSphere Business Modeler – The 2006 BPMS Report: IBM WebSphere BPM Suite v6.0, 2005. `ftp://ftp.software.ibm.com/software/integration/wbimodeler/library/whitepapers/bpms-ibm.pdf`.

[122] Eclipse BPEL Project. URL `http://www.eclipse.org/bpel/`.

[123] Mark Taber. Active Endpoints - ActiveVOS Quick Intro. URL `http://www.activevos.com/indepth/a_startHere/a_activeVOSaQuickIntro/QuickIntro.pdf`.

[124] OutSystems Platform. URL `http://express.outsystems.com/site/PlatformHome.aspx`.

[125] Michael Azoff. OutSystems Platform 4.0 – Technology Audit, Juni 2007.

[126] LabVIEW. URL `http://www.ni.com/labview/d/`.

[127] Particls. URL `http://www.particls.com/`.

[128] Yahoo Pipes. URL `http://pipes.yahoo.com/pipes/`.

[129] Macro.scopia. URL `http://macro.scopia.es/html/cpanel/home.html`.

[130] Art Baker and Jerry Lozano. *Windows® 2000 Device Driver Book: A Guide for Programmers, The Second Edition.* Prentice Hall, 2000.

[131] R. Meier and V. Cahill. Taxonomy of distributed event-based programming systems. In *22nd International Conference on Distributed Computing Systems Workshops*, 2002.

[132] Andreas Bobek, Hendrik Bohn, and Frank Golatowski. Voice-based Generic UPnP Control Point. In *2nd IEEE International Conference on Industrial Informatics, INDIN 04*, pages 487–492, Juni 2004.

[133] R. A. van Engelen. gSOAP – SOAP C++ Web Services. URL `http://www.cs.fsu.edu/~engelen/soap.html`.

[134] Apache Axis2. URL `http://ws.apache.org/axis2/`.

[135] The Apache Software Foundation. URL `http://www.apache.org/`.

[136] J.C. Gregorio. *URI Template.* Network Working Group, Internet Draft, 2006. `http://bitworking.org/projects/URI-Templates/draft-gregorio-uritemplate-00.html`.

[137] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman, Amsterdam, März 1995.

[138] P. Steggles and S. Gschwind. The Ubisense Smart Space Platform – A Ubisense White Paper, Mai 2005. `http://www.ubisense.de/media/pdf/Ubisense_a_smart_space_platform_1.pdf`.

[139] Jeffrey Hightower and Gaetano Borriello. A Survey and Taxonomy of Location Systems for Ubiquitous Computing, August 2001.

[140] Andreas Bobek, Elmar Zeeb, Frank Golatowski, and Dirk Timmermann. Entwicklung standardisierter Geräte mittels Geräte- und Dienstvorlagen für das Devices Profile for Web Services. In *12. Maritimes Symposium Elektrotechnik, Elektronik und Informationstechnik*, pages 185–190, Oktober 2007.

[141] Andreas Bobek, Elmar Zeeb, Hendrik Bohn, Frank Golatowski, and Dirk Timmermann. Device and Service Templates for the Devices Profile for Web Services. In *6th IEEE International Conference on Industrial Informatics (INDIN 08)*, Juli 2008.

[142] Kevin Mills and Christopher Dabrowski. Adaptive Jitter Control for UPnP M-Search. In *IEEE International Conference on Communications, ICC '03*, volume 2, pages 1008–1013, Mai 2003.

[143] P. Vixie, S. Thomson, Y. Rekhter, and J. Bound. *Dynamic Updates in the Domain Name System (DNS UPDATE)*. Request for Comments, April 1997. `http://www.ietf.org/rfc/rfc2136.txt`.

[144] Stuart Cheshire, Marc Krochmal, and Kiren Sekar. *Dynamic DNS Update Leases*. Internet-Draft, August 2006. `http://files.dns-sd.org/draft-sekar-dns-ul.txt`.

[145] Stuart Cheshire, Marc Krochmal, and Kiren Sekar. *DNS Long-Lived Queries (DNS-LLQ)*. Internet Draft, August 2006. `http://files.dns-sd.org/draft-sekar-dns-llq.txt`.

[146] Hung ying Tyan. Design, Realization and Evaluation of a Component-based Compositional Software Architecture for Network Simulation. Dissertation, The Ohio State University, 2002.

[147] J-Sim Home Page. URL `http://www.j-sim.org/`.

[148] Frank Reichenbach, Andreas Bobek, Philipp Hagen, and Dirk Timmermann. Increasing Lifetime of Wireless Sensor Networks with Energy-Aware Role-Changing. In *2nd IEEE International Workshop on Self-Managed Networks, Systems & Services (SelfMan 2006)*, pages 157–170, Juni 2006.

[149] Andre Bottaro. *RFP 86 – DPWS Discovery Base Driver*. OSGi Alliance, Mai 2007. `http://pagesperso-orange.fr/andre.bottaro/papers/rfp-86-DPWSDiscoveryBaseDriver.pdf`.

[150] André Bottaro, Eric Simon, Stéphane Seyvoz, and Anne Gérodolle. Dynamic Web Services on a Home Service Platform. *Advanced Information Networking and Applications, 2008. AINA 2008. 22nd International Conference on*, pages 378–385, March 2008.

[151] Amigo: Ambient intelligence for the networked home environment. URL `http://www.hitech-projects.com/euprojects/amigo/`.

A Anhang – XML-Schemas für DPWS Geräte- und Diensttypen

Listing A.1: XML-Schema für Dienst- und Gerätetypisierung

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://www.ws4d.org/templates/"
    targetNamespace="http://www.ws4d.org/templates/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    elementFormDefault="qualified">

    <import namespace="http://schemas.xmlsoap.org/wsdl/"
        schemaLocation="http://schemas.xmlsoap.org/wsdl/">
    </import>

    <complexType name="QNameType">
        <sequence>
            <element name="URI" type="anyURI"></element>
            <element name="localName" type="string"></element>
        </sequence>
    </complexType>

    <complexType name="ServiceType">
        <choice>
            <element name="Reference" type="anyURI"></element>
            <element ref="wsdl:definitions"></element>
        </choice>
    </complexType>

    <complexType name="HostedType">
        <sequence>
            <element name="Type" type="tns:QNameType"></element>
            <element name="Includes" type="tns:QNameType"
                minOccurs="0" maxOccurs="unbounded">
            </element>
            <element name="URLTemplate" type="string" minOccurs="0"></element>
            <element name="Service" type="tns:ServiceType"></element>
            <element name="ServiceId" type="string" minOccurs="0"></element>
            <element name="PortTypes" type="string" minOccurs="0"></element>
        </sequence>
    </complexType>

    <element name="Hosted" type="tns:HostedType"></element>

    <complexType name="HostType">
```

```
42          <sequence>
43              <element name="Type" type="tns:QNameType"></element>
44              <element name="Includes" type="tns:QNameType"
45                  minOccurs="0" maxOccurs="unbounded">
46              </element>
47              <element name="URLTemplate" type="string" minOccurs="0"></element>
48              <element name="ServiceId" type="string" minOccurs="0"></element>
49          </sequence>
50      </complexType>
51
52      <complexType name="RelationshipType">
53          <sequence>
54              <element name="Host" type="tns:HostType"></element>
55              <element name="HostedRef" type="anyURI" maxOccurs="unbounded">
56              </element>
57          </sequence>
58      </complexType>
59
60      <element name="Relationship" type="tns:RelationshipType"></element>
61
62  </schema>
```

B Anhang – Spezifikationen der DPWS-Lokalisierungssysteme und -dienste

B.1 Typdokumente der Lokalisierungssysteme

Listing B.1: ScanService-Diensttyp

```
<?xml version="1.0" encoding="UTF-8"?>
<t:HostedService xmlns:t="http://www.ws4d.org/templates/">
    <t:Type>
        <t:URI>http://www.ws4d.org/bluescan</t:URI>
        <t:localName>BlueScanScanService</t:localName>
    </t:Type>
    <t:PortType>
        <t:URI>http://www.ws4d.org/bluescan</t:URI>
        <t:localName>BlueScanScanPortType</t:localName>
    </t:PortType>
    <t:WsdlReference>
        http://www.ws4d.org/specs/bluescan/bluescan-scan-service.wsdl
    </t:WsdlReference>
</t:HostedService>
```

Listing B.2: EventService-Diensttyp

```
<?xml version="1.0" encoding="UTF-8"?>
<t:HostedService xmlns:t="http://www.ws4d.org/templates/">
    <t:Type>
        <t:URI>http://www.ws4d.org/bluescan</t:URI>
        <t:localName>BlueScanEventService</t:localName>
    </t:Type>
    <t:PortType>
        <t:URI>http://www.ws4d.org/bluescan</t:URI>
        <t:localName>BlueScanEventPortType</t:localName>
    </t:PortType>
    <t:WsdlReference>
        http://www.ws4d.org/specs/bluescan/bluescan-event-service.wsdl
    </t:WsdlReference>
</t:HostedService>
```

Listing B.3: BlueScan-Gerätetyp

```
<?xml version="1.0" encoding="UTF-8"?>
<t:Relationship xmlns:t="http://www.ws4d.org/templates/">
```

```
3     <t:Host>
4         <t:Type>
5             <t:URI>http://www.ws4d.org/bluescan</t:URI>
6             <t:localName>BlueScanDevice1.0</t:localName>
7         </t:Type>
8     </t:Host>
9     <t:Hosted>
10        <t:Reference>
11            http://www.ws4d.org/specs/bluescan/bluescan-scan-service-template.xml
12        </t:Reference>
13        <t:ServiceId>
14            http://www.ws4d.org/bluescan/scanservice
15        </t:ServiceId>
16    </t:Hosted>
17</t:Relationship>
```

Listing B.4: BlueScan-Gerätetyp der 2. Generation

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <t:Relationship xmlns:t="http://www.ws4d.org/templates/">
3     <t:Host>
4         <t:Type>
5             <t:URI>http://www.ws4d.org/bluescan</t:URI>
6             <t:localName>BlueScanDevice2.0</t:localName>
7         </t:Type>
8         <t:Includes>
9             <t:URI>http://www.ws4d.org/bluescan</t:URI>
10            <t:localName>BlueScanDevice1.0</t:localName>
11        </t:Includes>
12    </t:Host>
13    <t:Hosted>
14        <t:Reference>
15            http://www.ws4d.org/specs/bluescan/bluescan-scan-service-template.xml
16        </t:Reference>
17        <t:ServiceId>
18            http://www.ws4d.org/bluescan/scanservice
19        </t:ServiceId>
20    </t:Hosted>
21    <t:Hosted>
22        <t:Reference>
23            http://www.ws4d.org/specs/bluescan/bluescan-event-service-template.xml
24        </t:Reference>
25        <t:ServiceId>
26            http://www.ws4d.org/bluescan/eventservice
27        </t:ServiceId>
28    </t:Hosted>
29 </t:Relationship>
```

B.2 WSDL-Dokumente der Lokalisierungssysteme und -dienste

Listing B.5: BlueScan-ScanService-WSDL

```
<?xml version="1.0" encoding="UTF-8"?>

<wsdl:definitions targetNamespace="http://www.ws4d.org/bluescan"
    xmlns:tns="http://www.ws4d.org/bluescan"
    xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wse="http://schemas.xmlsoap.org/ws/2004/08/eventing"
    xmlns:wsoap12="http://schemas.xmlsoap.org/wsdl/soap12/">

    <wsdl:types>
        <xs:schema targetNamespace="http://www.ws4d.org/bluescan"
            xmlns:tns="http://www.ws4d.org/bluescan"
            xmlns:xs="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified" blockDefault="#all">

            <xs:complexType name="BlueScanConfigType">
                <xs:sequence>
                    <xs:element name="InquiryInterval" type="xs:integer"
                        minOccurs="1" maxOccurs="1">
                    </xs:element>
                    <xs:element name="Block" type="xs:integer"
                        minOccurs="0" maxOccurs="1">
                    </xs:element>
                </xs:sequence>
                <xs:any minOccurs="0" maxOccurs="unbounded"
                    namespace="##other" processContents="lax" />
            </xs:complexType>

            <xs:element name="BlueScanConfig"
                type="tns:BlueScanConfigType">
            </xs:element>

            <xs:complexType name="BlueScanResultType">
                <xs:sequence>
                    <xs:element name="InquiryTime" type="xs:dateTime"
                        minOccurs="1" maxOccurs="1">
                    </xs:element>
                    <xs:element name="BluetoothAddress" type="xs:string"
                        minOccurs="0" maxOccurs="unbounded">
                    </xs:element>
                </xs:sequence>
            </xs:complexType>

            <xs:element name="BlueScanResult"
                type="tns:BlueScanResultType">
            </xs:element>
```

```
47
48          </xs:schema>
49      </wsdl:types>
50
51      <wsdl:message name="SetConfigMessageIn">
52          <wsdl:part name="parameters" element="tns:BlueScanConfig" />
53      </wsdl:message>
54      <wsdl:message name="SetConfigMessageOut">
55          <wsdl:part name="parameters" element="tns:BlueScanConfig" />
56      </wsdl:message>
57      <wsdl:message name="ScanResultMessageOut">
58          <wsdl:part name="parameters" element="tns:BlueScanResult" />
59      </wsdl:message>
60
61      <wsdl:portType name="BlueScanScanPortType" wse:EventSource="true">
62          <wsdl:operation name="SetConfig">
63              <wsdl:input message="tns:SetConfigMessageIn"
64                  wsa:Action="http://www.ws4d.org/bluescan/BlueScanScanPortType
65                  /SetConfigMessageIn" />
66              <wsdl:output message="tns:SetConfigMessageOut"
67                  wsa:Action="http://www.ws4d.org/bluescan/BlueScanScanPortType
68                  /SetConfigMessageOut" />
69          </wsdl:operation>
70          <wsdl:operation name="ScanResult">
71              <wsdl:output message="tns:ScanResultMessageOut"
72                  wsa:Action="http://www.ws4d.org/bluescan/BlueScanScanPortType
73                  /ScanResultMessageOut" />
74          </wsdl:operation>
75      </wsdl:portType>
76
77  </wsdl:definitions>
```

Listing B.6: BlueScan-EventService-WSDL

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <wsdl:definitions targetNamespace="http://www.ws4d.org/bluescan"
4      xmlns:tns="http://www.ws4d.org/bluescan"
5      xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
6      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
7      xmlns:wse="http://schemas.xmlsoap.org/ws/2004/08/eventing"
8      xmlns:wsoap12="http://schemas.xmlsoap.org/wsdl/soap12/">
9
10     <wsdl:types>
11         <xs:schema targetNamespace="http://www.ws4d.org/bluescan"
12             xmlns:tns="http://www.ws4d.org/bluescan"
13             xmlns:xs="http://www.w3.org/2001/XMLSchema"
14             elementFormDefault="qualified" blockDefault="#all">
15
16             <xs:complexType name="TargetType">
```

```
17                  <xs:sequence>
18                      <xs:element name="BluetoothAddress" type="xs:string"
19                          minOccurs="1" maxOccurs="1">
20                      </xs:element>
21                      <xs:element name="ResponseTime" type="xs:dateTime"
22                          minOccurs="1" maxOccurs="1">
23                      </xs:element>
24                  </xs:sequence>
25              </xs:complexType>
26
27              <xs:complexType name="TargetListType">
28                  <xs:sequence>
29                      <xs:element name="Target" type="tns:TargetType"
30                          minOccurs="0" maxOccurs="unbounded">
31                      </xs:element>
32                  </xs:sequence>
33                  <xs:any minOccurs="0" maxOccurs="unbounded"
34                      namespace="##other" processContents="lax" />
35              </xs:complexType>
36
37              <xs:element name="TargetList" type="tns:TargetListType">
38              </xs:element>
39
40          </xs:schema>
41      </wsdl:types>
42
43      <wsdl:message name="InquiryEventMessageOut">
44          <wsdl:part name="parameters" element="tns:TargetList" />
45      </wsdl:message>
46      <wsdl:message name="ScanEventMessageOut">
47          <wsdl:part name="parameters" element="tns:TargetList" />
48      </wsdl:message>
49
50      <wsdl:portType name="BlueScanEventPortType"
51          wse:EventSource="true">
52          <wsdl:operation name="InquiryEvent">
53              <wsdl:output message="tns:InquiryEventMessageOut"
54                  wsa:Action="http://www.ws4d.org/bluescan/BlueScanEventPortType
55                  /InquiryEventMessageOut" />
56          </wsdl:operation>
57          <wsdl:operation name="ScanEvent">
58              <wsdl:output message="tns:ScanEventMessageOut"
59                  wsa:Action="http://www.ws4d.org/bluescan/BlueScanEventPortType
60                  /ScanEventMessageOut" />
61          </wsdl:operation>
62      </wsdl:portType>
63
64  </wsdl:definitions>
```

Listing B.7: BlueScan-Server-WSDL

```
<?xml version="1.0" encoding="UTF-8"?>

<wsdl:definitions targetNamespace="http://www.ws4d.org/bluescan"
    xmlns:tns="http://www.ws4d.org/bluescan"
    xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsoap12="http://schemas.xmlsoap.org/wsdl/soap12/">

    <wsdl:types>
        <xs:schema targetNamespace="http://www.ws4d.org/bluescan"
            xmlns:tns="http://www.ws4d.org/bluescan"
            xmlns:xs="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified" blockDefault="#all">

            <xs:complexType name="FindParamType">
                <xs:sequence>
                    <xs:element name="BluetoothAddress" type="xs:string"
                        minOccurs="1" maxOccurs="1">
                    </xs:element>
                    <xs:element name="OffsetTime" type="xs:duration"
                        minOccurs="1" maxOccurs="1">
                    </xs:element>
                </xs:sequence>
            </xs:complexType>

            <xs:element name="FindParam" type="tns:FindParamType">
            </xs:element>

            <xs:complexType name="FindResultType">
                <xs:sequence>
                    <xs:element name="Location" type="xs:string"
                        minOccurs="0" maxOccurs="unbounded">
                    </xs:element>
                </xs:sequence>
            </xs:complexType>

            <xs:element name="FindResult" type="tns:FindResultType">
            </xs:element>

            <xs:complexType name="ScanParamType">
                <xs:sequence>
                    <xs:element name="Location" type="xs:string"
                        minOccurs="1" maxOccurs="1">
                    </xs:element>
                    <xs:element name="OffsetTime" type="xs:duration"
                        minOccurs="1" maxOccurs="1">
                    </xs:element>
                </xs:sequence>
            </xs:complexType>
```

```
50
51              <xs:element name="ScanParam" type="tns:ScanParamType">
52              </xs:element>
53
54              <xs:complexType name="TargetType">
55                  <xs:sequence>
56                      <xs:element name="BluetoothAddress" type="xs:string"
57                          minOccurs="1" maxOccurs="1">
58                      </xs:element>
59                      <xs:element name="ResponseTime" type="xs:dateTime"
60                          minOccurs="1" maxOccurs="1">
61                      </xs:element>
62                  </xs:sequence>
63              </xs:complexType>
64
65              <xs:complexType name="ScanResultType">
66                  <xs:sequence>
67                      <xs:element name="Target" type="tns:TargetType"
68                          minOccurs="0" maxOccurs="unbounded">
69                      </xs:element>
70                  </xs:sequence>
71              </xs:complexType>
72
73              <xs:element name="ScanResult" type="tns:ScanResultType">
74              </xs:element>
75
76          </xs:schema>
77      </wsdl:types>
78
79      <wsdl:message name="FindMessageIn">
80          <wsdl:part name="parameters" element="tns:FindParam" />
81      </wsdl:message>
82      <wsdl:message name="FindMessageOut">
83          <wsdl:part name="parameters" element="tns:FindResult" />
84      </wsdl:message>
85      <wsdl:message name="ScanMessageIn">
86          <wsdl:part name="parameters" element="tns:ScanParam" />
87      </wsdl:message>
88      <wsdl:message name="ScanMessageOut">
89          <wsdl:part name="parameters" element="tns:ScanResult" />
90      </wsdl:message>
91
92      <wsdl:portType name="BlueScanServerPortType">
93          <wsdl:operation name="Find">
94              <wsdl:input message="tns:FindMessageIn"
95                  wsa:Action="http://www.ws4d.org/bluescan
96                  /BlueScanServerPortType/FindMessageIn" />
97              <wsdl:output message="tns:FindMessageOut"
98                  wsa:Action="http://www.ws4d.org/bluescan
99                  /BlueScanServerPortType/FindMessageOut" />
```

```
100         </wsdl:operation>
101         <wsdl:operation name="Scan">
102             <wsdl:input message="tns:ScanMessageIn"
103                 wsa:Action="http://www.ws4d.org/bluescan
104                 /BlueScanServerPortType/ScanMessageIn" />
105             <wsdl:output message="tns:ScanMessageOut"
106                 wsa:Action="http://www.ws4d.org/bluescan
107                 /BlueScanServerPortType/ScanMessageOut" />
108         </wsdl:operation>
109     </wsdl:portType>
110
111 </wsdl:definitions>
```

Listing B.8: Ubisense-Gateway-WSDL

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <wsdl:definitions targetNamespace="http://www.ws4d.org/ubisense"
4      xmlns:tns="http://www.ws4d.org/ubisense"
5      xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
6      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
7      xmlns:wse="http://schemas.xmlsoap.org/ws/2004/08/eventing"
8      xmlns:wsoap12="http://schemas.xmlsoap.org/wsdl/soap12/">
9
10     <wsdl:types>
11         <xs:schema targetNamespace="http://www.ws4d.org/ubisense"
12             xmlns:tns="http://www.ws4d.org/ubisense"
13             xmlns:xs="http://www.w3.org/2001/XMLSchema"
14             elementFormDefault="qualified" blockDefault="#all">
15
16             <xs:complexType name="PointType">
17                 <xs:sequence>
18                     <xs:element name="X" type="xs:double" minOccurs="1"
19                         maxOccurs="1">
20                     </xs:element>
21                     <xs:element name="Y" type="xs:double" minOccurs="1"
22                         maxOccurs="1">
23                     </xs:element>
24                     <xs:element name="Z" type="xs:double" minOccurs="1"
25                         maxOccurs="1">
26                     </xs:element>
27                 </xs:sequence>
28             </xs:complexType>
29
30             <xs:complexType name="TagType">
31                 <xs:sequence>
32                     <xs:element name="ID" type="xs:string" minOccurs="1"
33                         maxOccurs="1">
34                     </xs:element>
35                     <xs:element name="Point" type="tns:PointType"
```

```
36                          minOccurs="0" maxOccurs="1">
37                      </xs:element>
38                  </xs:sequence>
39              </xs:complexType>
40
41              <xs:complexType name="TagListType">
42                  <xs:sequence>
43                      <xs:element name="Tag" type="tns:TagType"
44                          minOccurs="0" maxOccurs="unbounded">
45                      </xs:element>
46                  </xs:sequence>
47              </xs:complexType>
48
49              <xs:element name="TagList" type="tns:TagListType"></xs:element>
50
51              <xs:complexType name="BoxType">
52                  <xs:sequence>
53                      <xs:element name="PointA" type="tns:PointType"
54                          minOccurs="1" maxOccurs="1">
55                      </xs:element>
56                      <xs:element name="PointB" type="tns:PointType"
57                          minOccurs="1" maxOccurs="1">
58                      </xs:element>
59                  </xs:sequence>
60              </xs:complexType>
61
62              <xs:element name="Box" type="tns:BoxType"></xs:element>
63
64          </xs:schema>
65      </wsdl:types>
66
67      <wsdl:message name="ReadTagsMessageIn">
68          <wsdl:part name="parameters" element="tns:TagList" />
69      </wsdl:message>
70      <wsdl:message name="ReadTagsMessageOut">
71          <wsdl:part name="parameters" element="tns:TagList" />
72      </wsdl:message>
73      <wsdl:message name="ReadBoxMessageIn">
74          <wsdl:part name="parameters" element="tns:Box" />
75      </wsdl:message>
76      <wsdl:message name="ReadBoxMessageOut">
77          <wsdl:part name="parameters" element="tns:TagList" />
78      </wsdl:message>
79      <wsdl:message name="BoxChangedMessageOut">
80          <wsdl:part name="parameters" element="tns:TagList" />
81      </wsdl:message>
82
83      <wsdl:portType name="UbisensePortType" wse:EventSource="true">
84          <wsdl:operation name="ReadTags">
85              <wsdl:input message="tns:ReadTagsMessageIn"
```

```
86                  wsa:Action="http://www.ws4d.org/ubisense/ReadTagsMessageIn"/>
87              <wsdl:output message="tns:ReadTagsMessageOut"
88                  wsa:Action="http://www.ws4d.org/ubisense/ReadTagsMessageOut"/>
89          </wsdl:operation>
90          <wsdl:operation name="ReadBox">
91              <wsdl:input message="tns:ReadBoxMessageIn"
92                  wsa:Action="http://www.ws4d.org/ubisense/ReadBoxMessageIn" />
93              <wsdl:output message="tns:ReadBoxMessageOut"
94                  wsa:Action="http://www.ws4d.org/ubisense/ReadBoxMessageOut" />
95          </wsdl:operation>
96          <wsdl:operation name="BoxChanged">
97              <wsdl:output message="tns:BoxChangedMessageOut"
98                  wsa:Action="http://www.ws4d.org/ubisense
99                  /BoxChangedMessageOut" />
100         </wsdl:operation>
101     </wsdl:portType>
102
103 </wsdl:definitions>
```

Listing B.9: Location-Service-WSDL

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <wsdl:definitions targetNamespace="http://www.ws4d.org/ls"
4      xmlns:tns="http://www.ws4d.org/ls"
5      xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
6      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
7      xmlns:wse="http://schemas.xmlsoap.org/ws/2004/08/eventing"
8      xmlns:wsoap12="http://schemas.xmlsoap.org/wsdl/soap12/">
9
10     <wsdl:types>
11         <xs:schema targetNamespace="http://www.ws4d.org/ls"
12             xmlns:tns="http://www.ws4d.org/ls"
13             xmlns:xs="http://www.w3.org/2001/XMLSchema"
14             elementFormDefault="qualified" blockDefault="#all">
15
16             <xs:complexType name="ObjectType">
17                 <xs:sequence>
18                     <xs:any minOccurs="1" maxOccurs="1"
19                         namespace="##other" processContents="lax" />
20                 </xs:sequence>
21                 <xs:attribute name="dialect" type="xs:anyURI"
22                     use="optional">
23                 </xs:attribute>
24             </xs:complexType>
25
26             <xs:complexType name="LocationType">
27                 <xs:sequence>
28                     <xs:any minOccurs="1" maxOccurs="1"
29                         namespace="##other" processContents="lax" />
```

```
30                  </xs:sequence>
31                  <xs:attribute name="dialect" type="xs:anyURI"
32                      use="optional">
33                  </xs:attribute>
34              </xs:complexType>
35
36              <xs:complexType name="QualifierType">
37                  <xs:sequence>
38                      <xs:any minOccurs="1" maxOccurs="1"
39                          namespace="##other" processContents="lax" />
40                  </xs:sequence>
41                  <xs:attribute name="dialect" type="xs:anyURI"
42                      use="optional">
43                  </xs:attribute>
44              </xs:complexType>
45
46              <xs:complexType name="QualityType">
47                  <xs:sequence>
48                      <xs:any minOccurs="1" maxOccurs="1"
49                          namespace="##other" processContents="lax" />
50                  </xs:sequence>
51                  <xs:attribute name="dialect" type="xs:anyURI"
52                      use="optional">
53                  </xs:attribute>
54              </xs:complexType>
55
56
57              <xs:complexType name="ObjectQualityType">
58                  <xs:sequence>
59                      <xs:element name="Object" type="tns:ObjectType"
60                          minOccurs="1" maxOccurs="1">
61                      </xs:element>
62                      <xs:element name="Quality" type="tns:QualityType"
63                          minOccurs="0" maxOccurs="1">
64                      </xs:element>
65                  </xs:sequence>
66              </xs:complexType>
67
68              <xs:complexType name="ObjectQualifierType">
69                  <xs:sequence>
70                      <xs:element name="Object" type="tns:ObjectType"
71                          minOccurs="1" maxOccurs="1">
72                      </xs:element>
73                      <xs:element name="Qualifier"
74                          type="tns:QualifierType" minOccurs="0" maxOccurs="1">
75                      </xs:element>
76                  </xs:sequence>
77              </xs:complexType>
78
79              <xs:complexType name="LocationQualityType">
```

```
80                  <xs:sequence>
81                      <xs:element name="Location" type="tns:LocationType"
82                          minOccurs="1" maxOccurs="1">
83                      </xs:element>
84                      <xs:element name="Quality" type="tns:QualityType"
85                          minOccurs="0" maxOccurs="1">
86                      </xs:element>
87                  </xs:sequence>
88              </xs:complexType>
89
90              <xs:complexType name="LocationQualifierType">
91                  <xs:sequence>
92                      <xs:element name="Location" type="tns:LocationType"
93                          minOccurs="1" maxOccurs="1">
94                      </xs:element>
95                      <xs:element name="Qualifier"
96                          type="tns:QualifierType" minOccurs="0" maxOccurs="1">
97                      </xs:element>
98                  </xs:sequence>
99              </xs:complexType>
100
101
102             <xs:complexType name="ObjectQualifierListType">
103                 <xs:sequence>
104                     <xs:element name="ObjectQualifier"
105                         type="tns:ObjectQualifierType" minOccurs="1"
106                         maxOccurs="unbounded">
107                     </xs:element>
108                 </xs:sequence>
109             </xs:complexType>
110
111             <xs:complexType name="LocationQualifierListType">
112                 <xs:sequence>
113                     <xs:element name="LocationQualifier"
114                         type="tns:LocationQualifierType" minOccurs="1"
115                         maxOccurs="unbounded">
116                     </xs:element>
117                 </xs:sequence>
118             </xs:complexType>
119
120
121             <xs:complexType name="ObjectLocationsType">
122                 <xs:sequence>
123                     <xs:element name="ObjectQualifier"
124                         type="tns:ObjectQualifierType" minOccurs="1"
125                         maxOccurs="1">
126                     </xs:element>
127                     <xs:element name="LocationQuality"
128                         type="tns:LocationQualityType" minOccurs="0"
129                         maxOccurs="unbounded">
```

```
130                     </xs:element>
131                 </xs:sequence>
132             </xs:complexType>
133
134             <xs:complexType name="LocationObjectsType">
135                 <xs:sequence>
136                     <xs:element name="LocationQualifier"
137                         type="tns:LocationQualifierType" minOccurs="1"
138                         maxOccurs="1">
139                     </xs:element>
140                     <xs:element name="ObjectQuality"
141                         type="tns:ObjectQualityType" minOccurs="0"
142                         maxOccurs="unbounded">
143                     </xs:element>
144                 </xs:sequence>
145             </xs:complexType>
146
147             <xs:complexType name="ObjectLocationsListType">
148                 <xs:sequence>
149                     <xs:element name="ObjectLocations"
150                         type="tns:ObjectLocationsType" minOccurs="1"
151                         maxOccurs="unbounded">
152                     </xs:element>
153                 </xs:sequence>
154             </xs:complexType>
155
156             <xs:complexType name="LocationObjectsListType">
157                 <xs:sequence>
158                     <xs:element name="LocationObjects"
159                         type="tns:LocationObjectsType" minOccurs="1"
160                         maxOccurs="unbounded">
161                     </xs:element>
162                 </xs:sequence>
163             </xs:complexType>
164
165             <xs:element name="LocateRequest"
166                 type="tns:ObjectQualifierListType">
167             </xs:element>
168
169             <xs:element name="LocateResponse"
170                 type="tns:ObjectLocationsListType">
171             </xs:element>
172
173             <xs:element name="MonitorRequest"
174                 type="tns:LocationQualifierListType">
175             </xs:element>
176
177             <xs:element name="MonitorResponse"
178                 type="tns:LocationObjectsListType">
179             </xs:element>
```

```
180
181         </xs:schema>
182     </wsdl:types>
183
184     <wsdl:message name="LocateMessageIn">
185         <wsdl:part name="parameters" element="tns:LocateRequest" />
186     </wsdl:message>
187     <wsdl:message name="LocateMessageOut">
188         <wsdl:part name="parameters" element="tns:LocateResponse" />
189     </wsdl:message>
190     <wsdl:message name="MonitorMessageIn">
191         <wsdl:part name="parameters" element="tns:MonitorRequest" />
192     </wsdl:message>
193     <wsdl:message name="MonitorMessageOut">
194         <wsdl:part name="parameters" element="tns:MonitorResponse" />
195     </wsdl:message>
196
197     <wsdl:portType name="LocationPortType">
198         <wsdl:operation name="Locate">
199             <wsdl:input message="tns:LocateMessageIn"
200                 wsa:Action="http://www.ws4d.org/ls/LocationPortType
201                 /LocateMessageIn" />
202             <wsdl:output message="tns:LocateMessageOut"
203                 wsa:Action="http://www.ws4d.org/ls/LocationPortType
204                 /LocateMessageOut" />
205         </wsdl:operation>
206         <wsdl:operation name="Monitor">
207             <wsdl:input message="tns:MonitorMessageIn"
208                 wsa:Action="http://www.ws4d.org/ls/LocationPortType
209                 /MonitorMessageIn" />
210             <wsdl:output message="tns:MonitorMessageOut"
211                 wsa:Action="http://www.ws4d.org/ls/LocationPortType
212                 /MonitorMessageOut" />
213         </wsdl:operation>
214     </wsdl:portType>
215
216 </wsdl:definitions>
```

C Anhang – Spezifikationen der WODA-Dienste

Listing C.1: Relevanter Auszug aus dem XML-Schema von DPWS für WODA-Geräte

```
<xs:schema
    targetNamespace="http://schemas.xmlsoap.org/ws/2006/02/devprof"
    xmlns:tns="http://schemas.xmlsoap.org/ws/2006/02/devprof"
    xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified"
    blockDefault="#all" >

    <xs:import
        namespace="http://schemas.xmlsoap.org/ws/2004/08/addressing"
        schemaLocation
        ="http://schemas.xmlsoap.org/ws/2004/08/addressing/addressing.xsd"
        />

    <xs:complexType name="LocalizedStringType" >
        <xs:simpleContent>
            <xs:extension base="xs:string" >
                <xs:anyAttribute namespace="##other" processContents="lax" />
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>

    <xs:complexType name="ThisModelType" >
        <xs:sequence>
            <xs:element name="Manufacturer" type="tns:LocalizedStringType"
                maxOccurs="unbounded" />
            <xs:element name="ManufacturerUrl" type="xs:anyURI"
                minOccurs="0" />
            <xs:element name="ModelName" type="tns:LocalizedStringType"
                maxOccurs="unbounded" />
            <xs:element name="ModelNumber" type="xs:string" minOccurs="0" />
            <xs:element name="ModelUrl" type="xs:anyURI" minOccurs="0" />
            <xs:element name="PresentationUrl" type="xs:anyURI"
                minOccurs="0" />
            <xs:any namespace="##other" processContents="lax"
                minOccurs="0" maxOccurs="unbounded" />
        </xs:sequence>
        <xs:anyAttribute namespace="##other" processContents="lax" />
    </xs:complexType>

    <xs:element name="ThisModel" type="tns:ThisModelType" />
```

```
42
43      <xs:complexType name="ThisDeviceType" >
44          <xs:sequence>
45              <xs:element name="FriendlyName" type="tns:LocalizedStringType"
46                  maxOccurs="unbounded" />
47              <xs:element name="FirmwareVersion" type="xs:string"
48                  minOccurs="0" />
49              <xs:element name="SerialNumber" type="xs:string" minOccurs="0" />
50              <xs:any namespace="##other" processContents="lax"
51                  minOccurs="0" maxOccurs="unbounded" />
52          </xs:sequence>
53          <xs:anyAttribute namespace="##other" processContents="lax" />
54      </xs:complexType>
55
56      <xs:element name="ThisDevice" type="tns:ThisDeviceType" />
57
58  </xs:schema>
```

Listing C.2: XML-Schema für Beschreibung von WODA-Geräten

```
1   <xs:schema targetNamespace="http://www.ws4d.org/woda/dev"
2       xmlns:tns="http://www.ws4d.org/woda/dev"
3       xmlns:dpws="http://schemas.xmlsoap.org/ws/2006/02/devprof"
4       xmlns:xs="http://www.w3.org/2001/XMLSchema"
5       elementFormDefault="qualified" blockDefault="#all">
6
7       <xs:import namespace="http://schemas.xmlsoap.org/ws/2006/02/devprof"
8           schemaLocation="http://schemas.xmlsoap.org/ws/2006/02/devprof
9           /devicesprofile.xsd" />
10
11      <xs:complexType name="ServiceCType">
12          <xs:sequence>
13              <xs:element name="InstanceName" type="xs:string"></xs:element>
14              <xs:element name="ServiceType" type="xs:string"></xs:element>
15              <xs:element name="Url" type="xs:anyURI" minOccurs="0" />
16              <xs:element name="Wadl" type="xs:anyURI" minOccurs="0" />
17          </xs:sequence>
18      </xs:complexType>
19
20      <xs:complexType name="ServicesType">
21          <xs:sequence>
22              <xs:element name="Service" type="tns:ServiceCType"
23                  minOccurs="0" maxOccurs="unbounded">
24              </xs:element>
25          </xs:sequence>
26      </xs:complexType>
27
28      <xs:complexType name="DeviceType">
29          <xs:sequence>
30              <xs:element ref="dpws:ThisModel"></xs:element>
```

```
            <xs:element ref="dpws:ThisDevice"></xs:element>
            <xs:element name="Services" type="tns:ServicesType">
            </xs:element>
        </xs:sequence>
    </xs:complexType>

    <xs:element name="Device" type="tns:DeviceType" />

</xs:schema>
```

Listing C.3: XML-Schema für den WODA-Ubisense-Dienst

```
<xs:schema targetNamespace="http://www.ws4d.org/ubisense"
    xmlns:tns="http://www.ws4d.org/ubisense"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" blockDefault="#all">

    <xs:complexType name="PointType">
        <xs:sequence>
            <xs:element name="X" type="xs:double" minOccurs="1"
                maxOccurs="1">
            </xs:element>
            <xs:element name="Y" type="xs:double" minOccurs="1"
                maxOccurs="1">
            </xs:element>
            <xs:element name="Z" type="xs:double" minOccurs="1"
                maxOccurs="1">
            </xs:element>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="TagType">
        <xs:sequence>
            <xs:element name="ID" type="xs:string" minOccurs="1"
                maxOccurs="1">
            </xs:element>
            <xs:element name="Point" type="tns:PointType"
                minOccurs="0" maxOccurs="1">
            </xs:element>
        </xs:sequence>
    </xs:complexType>

    <xs:element name="Tag" type="tns:TagType"></xs:element>

    <xs:complexType name="TagListType">
        <xs:sequence>
            <xs:element ref="tns:Tag" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>

```

```
39      <xs:element name="TagList" type="tns:TagListType"></xs:element>
40
41      <xs:complexType name="BoxType">
42          <xs:sequence>
43              <xs:element name="PointA" type="tns:PointType"
44                  minOccurs="1" maxOccurs="1">
45              </xs:element>
46              <xs:element name="PointB" type="tns:PointType"
47                  minOccurs="1" maxOccurs="1">
48              </xs:element>
49          </xs:sequence>
50      </xs:complexType>
51
52      <xs:element name="Box" type="tns:BoxType"></xs:element>
53
54      <xs:complexType name="RangeType">
55          <xs:sequence>
56              <xs:element name="Box" type="tns:BoxType"
57                  minOccurs="1" maxOccurs="1">
58              </xs:element>
59              <xs:element name="TagList" type="tns:TagListType"
60                  minOccurs="0" maxOccurs="1">
61              </xs:element>
62          </xs:sequence>
63      </xs:complexType>
64
65      <xs:element name="Range" type="tns:RangeType"></xs:element>
66
67      <xs:element name="Ranges">
68          <xs:complexType>
69              <xs:sequence>
70                  <xs:element ref="tns:Range" minOccurs="0"
71                              maxOccurs="unbounded"/>
72              </xs:sequence>
73          </xs:complexType>
74      </xs:element>
75
76  </xs:schema>
```

Listing C.4: WADL-Dokument für den WODA-Ubisense-Dienst

```
1   <application xmlns="http://research.sun.com/wadl/2006/07"
2       xmlns:ub="http://www.ws4d.org/ubisense"
3       xmlns:xs="http://www.w3.org/2001/XMLSchema">
4
5       <grammars>
6           <include href="ubisense.xsd"/>
7       </grammars>
8
9       <resources base="http://ubisense.local:5100/">
```

```
10          <resource path="tags">
11              <method name="GET">
12                  <response>
13                      <representation mediaType="text/xml"
14                                      element="ub:TagList"/>
15                  </response>
16              </method>
17          </resource>
18          <resource path="tag/{id}">
19              <param name="id" style="template" type="xs:string"/>
20              <method name="GET">
21                  <response>
22                      <representation mediaType="text/xml" element="ub:Tag"/>
23                  </response>
24              </method>
25          </resource>
26          <resource path="ranges">
27              <method name="GET">
28                  <response>
29                      <representation mediaType="text/xml" element="ub:Ranges"/>
30                  </response>
31              </method>
32              <method name="POST">
33                  <request>
34                      <representation mediaType="text/xml" element="ub:Range"/>
35                  </request>
36                  <response>
37                      <doc>see URI in 201 Created Header</doc>
38                  </response>
39              </method>
40          </resource>
41          <resource path="range/{id}">
42              <param name="id" style="template" type="xs:string"/>
43              <method name="GET">
44                  <response>
45                      <representation mediaType="text/xml" element="ub:Range"/>
46                  </response>
47              </method>
48              <method name="PUT">
49                  <request>
50                      <representation mediaType="text/xml" element="ub:Range"/>
51                  </request>
52              </method>
53              <method name="DELETE">
54              </method>
55              <method name="MIRROR">
56              </method>
57          </resource>
58      </resources>
59  </application>
```

Listing C.5: WODA-Beschreibung für das Ubisense-Gerät

```
<woda:Device xmlns:woda="http://www.ws4d.org/woda/dev"
    xmlns:dpws="http://schemas.xmlsoap.org/ws/2006/02/devprof">
    <woda:Services>
        <woda:Service>
            <woda:InstanceName>Labor</woda:InstanceName>
            <woda:ServiceType>_ubisense._woda._http._tcp</woda:ServiceType>
            <woda:Url>http://ubisense.local:5100/</woda:Url>
            <woda:Wadl>http://ubisense.local:5100/ubisense.wadl</woda:Wadl>
        </woda:Service>
        <woda:Service>
            <woda:InstanceName>Labor</woda:InstanceName>
            <woda:ServiceType>_power._woda._http._tcp</woda:ServiceType>
            <woda:Url>http://ubisense.local:4400/</woda:Url>
        </woda:Service>
    </woda:Services>
    <dpws:ThisModel>
        <dpws:Manufacturer>Ubisense</dpws:Manufacturer>
        <dpws:ManufacturerUrl>http://www.ubisense.de/</dpws:ManufacturerUrl>
        <dpws:ManufacturerUrl>http://www.ubisense.net/</dpws:ManufacturerUrl>
        <dpws:ModelName>Ubisense Serie 7000</dpws:ModelName>
        <dpws:PresentationUrl>http://ubisense.local:5100/web
        </dpws:PresentationUrl>
    </dpws:ThisModel>
    <dpws:ThisDevice>
        <dpws:FriendlyName>Ubisense Labor Uni Rostock</dpws:FriendlyName>
        <dpws:SerialNumber>23498-25874698-2</dpws:SerialNumber>
    </dpws:ThisDevice>
</woda:Device>
```

D Anhang – Pipes

Listing D.1: Beispiel für ein Pipes-Dokument

```
{
    "wires": [
        {
            "portA": "point",
            "portB": "point",
            "moduleB": 2,
            "ipoa": true,
            "moduleA": 0
        },
        {
            "portA": "x",
            "portB": "valuePort3",
            "moduleB": 3,
            "ipoa": true,
            "moduleA": 2
        },
        {
            "portA": "y",
            "portB": "valuePort4",
            "moduleB": 3,
            "ipoa": true,
            "moduleA": 2
        },
        {
            "portA": "out",
            "portB": "tag",
            "moduleB": 0,
            "ipoa": true,
            "moduleA": 1
        },
        {
            "portA": "out",
            "portB": "valuePort2",
            "moduleB": 3,
            "ipoa": false,
            "moduleA": 1
        },
        {
            "portA": "result",
            "portB": "feedIn",
            "moduleB": 4,
```

```
42              "ipoa": true,
43              "moduleA": 3
44          }
45      ],
46      "modules": [
47          {
48              "ports": [
49                  {
50                      "name": "point",
51                      "type": "out"
52                  },
53                  {
54                      "name": "url",
55                      "type": "in"
56                  },
57                  {
58                      "name": "tag",
59                      "type": "in"
60                  }
61              ],
62              "name": 1212973730779,
63              "state": {
64                  "tag": "",
65                  "url": "http://192.168.2.10:5293/ubisense"
66              },
67              "target": 0,
68              "type": "ubisense",
69              "y": 104,
70              "x": 742
71          },
72          {
73              "ports": [{
74                  "name": "out",
75                  "type": "out"
76              }],
77              "name": 1213013735448,
78              "state": {"text": "62490-125"},
79              "target": 1,
80              "type": "text.source",
81              "y": 175,
82              "x": 526
83          },
84          {
85              "ports": [
86                  {
87                      "name": "point",
88                      "type": "in"
89                  },
90                  {
91                      "name": "x",
```

```
                    "type": "out"
                },
                {
                    "name": "y",
                    "type": "out"
                },
                {
                    "name": "z",
                    "type": "out"
                }
            ],
            "name": 1213013776643,
            "state": {},
            "target": 2,
            "type": "pointsplitter",
            "y": 277,
            "x": 672
        },
        {
            "ports": [
                {
                    "name": "result",
                    "type": "out"
                },
                {
                    "name": "keyPort1",
                    "type": "in"
                },
                {
                    "name": "valuePort1",
                    "type": "in"
                },
                {
                    "name": "keyPort2",
                    "type": "in"
                },
                {
                    "name": "valuePort2",
                    "type": "in"
                },
                {
                    "name": "keyPort3",
                    "type": "in"
                },
                {
                    "name": "valuePort3",
                    "type": "in"
                },
                {
                    "name": "keyPort4",
```

```
142                     "type": "in"
143                 },
144                 {
145                     "name": "valuePort4",
146                     "type": "in"
147                 }
148             ],
149             "name": 1213013785379,
150             "state": {"keyValues": [
151                 {
152                     "keyString": "name",
153                     "id": 1,
154                     "valueString": "Andreas"
155                 },
156                 {
157                     "keyString": "tag",
158                     "id": 2,
159                     "valueString": ""
160                 },
161                 {
162                     "keyString": "x",
163                     "id": 3,
164                     "valueString": ""
165                 },
166                 {
167                     "keyString": "y",
168                     "id": 4,
169                     "valueString": ""
170                 }
171             ]},
172             "target": 3,
173             "type": "jsonbuilder",
174             "y": 267,
175             "x": 361
176         },
177         {
178             "ports": [
179                 {
180                     "name": "feedIn",
181                     "type": "in"
182                 },
183                 {
184                     "name": "resultOut",
185                     "type": "out"
186                 }
187             ],
188             "name": 1213014505691,
189             "state": {"template": "{name} ({tag}) is located at ({x},{y})"},
190             "target": 4,
191             "type": "substitution",
```

```
            "y": 448,
            "x": 572
        }
    ]
}
```

Erklärung

Ich versichere, diese Dissertation selbstständig und ohne fremde Hilfe angefertigt und die benutzten Quellen und Hilfsmittel vollständig angegeben zu haben.

Ort, Datum: Rostock, 09.07.2008 Andreas Bobek

Thesen

Ubiquitous Computing und Frameworks

1. Aus Hardwaresicht sind die Voraussetzungen gegeben, um die Vision des Ubiquitous Computings umzusetzen.

2. Um Ubiquitous Computing zu realisieren, ist ein universell einsetzbares Framework erforderlich, welches Protokolle, Methoden, Werkzeuge und eine Laufzeitumgebung umfasst, die für die Entwicklung und den Betrieb von Geräten und Diensten benötigt werden.

3. Universell einsetzbare Frameworks müssen unabhängig von Hardware, Plattformen, Programmiersprachen, Programmierkonzepten und Domänen funktionieren. Außerdem müssen sie internetfähig sein und sollten Merkmale serviceorientierter Architekturen (SOA) aufweisen. Die meisten (fast alle) existierenden Lösungen erfüllen mindestens eine dieser Anforderungen nicht.

4. Universal Plug and Play (UPnP) verwendet proprietäre und teilweise schlecht entworfene Protokolle, besitzt jedoch mit den Device Control Protocols (DCP) gerätespezifische Anwendungsprotokolle, die in der Praxis zu interoperablen Lösungen führen.

5. Das Devices Profile for Web Services (DPWS) ist noch sehr jung und bisher kaum verbreitet, verfügt jedoch aufgrund der Nähe zu einer Vielzahl anderer Web-Services-Protokolle über ein hohes Akzeptanzpotenzial.

Web Services und das Devices Profile for Web Services

6. Das Devices Profile for Web Services (DPWS) wurde für den auf Java basierenden Axis2-SOAP-Prozessor implementiert. Axis2 verwendet eine modulare Architektur, mit welcher Web-Services-Spezifikationen durch Module umgesetzt werden können.

7. Für die in DPWS verwendeten Protokolle WS-Discovery, WS-Eventing, WS-Transfer, WS-MetadataExchange und für DPWS wurden jeweils Axis2-Module implementiert. Damit folgen Axis2 sowie die DPWS-Umsetzung dem Ansatz, Web-Services-Protokolle in Form

vorgefertigter Softwarekomponenten anderen Diensten und Anwendungen zur Verfügung zu stellen.

8. Mit der DPWS-Umsetzung für Axis2 ist es erstmals möglich DPWS-Geräte in Enterprise-Systemen, in denen Axis2 eingesetzt wird, zu verwenden.

9. DPWS fehlt ein Typisierungsmechanismus für Geräte und Dienste. Dieser ist für Hersteller und Entwickler jedoch wichtig, um Interoperabilität zu erzielen und um Code für die Implementierungen automatisch generierbar zu machen.

10. Der eigene Typisierungsansatz basiert auf XML, unterstützt Versionierung und ist für die automatische Codegenerierung geeignet. Durch die Verwendung von URL-Templates kann die Discovery-Phase deutlich verkürzt werden.

11. Mit der Entwicklung einer eigenen Lokalisierungsplattform wurde gezeigt, dass DPWS für den evolutionären Aufbau einer SOA-basierten IT-Infrastruktur geeignet ist. Neben DPWS wurden dazu Konzepte von XML-Schema und der eigene Typisierungsmechanismus ausgenutzt.

12. DPWS verwendet eine Vielzahl von Protokollen, die derzeit immer noch großen Änderungen und Anpassungen unterliegen. Daher wird DPWS zukünftig ebenfalls noch aktualisiert werden.

13. Eine große Gefahr für DPWS besteht in den Web Services selbst: Die ursprüngliche Erwartung, dass durch den offenen Spezifikationsprozess schnell interoperable Protokolle verabschiedet werden können, die die unterschiedlichen Anforderungen vieler Anwender berücksichtigen, hat sich nicht erfüllt. Stattdessen weisen die Protokolle Widersprüche und Redundanzen auf.

14. Web Services benötigen noch mehrere Jahre, bis sie transparent, also für den Anwender nicht mehr sichtbar, benutzt werden können.

Die Web-oriented Device Architecture

15. Die Web-oriented Device Architecture (WODA) ist ein eigener, alternativer Ansatz, um ein geräteintegrierendes Framework umzusetzen. Die architektonische Grundlage bildet der Representational State Transfer (REST). Als Plattform wird das existierende Internet verwendet.

16. WODA ist ein universell anwendbares Framework, da es mit DNS, HTTP und Ajax etablierte Protokolle und Technologien verwendet, die global verfügbar sind. Fehlende Komponenten wie die Beschreibung von Geräten und Diensten sowie ein Ereignismechanismus wurden so ergänzt, dass sie REST-konform und zusammen mit den anderen Ansätzen funktionieren.

17. Neben der automatischen Dienstnutzung ist es wichtig, Geräte und Dienste durch Benutzer visuell konfigurieren, überwachen und steuern zu können. Für diese Aufgabe ist ein herkömmlicher Webbrowser sehr gut geeignet, da er auf vielen Plattformen verfügbar ist und sich mit HTML leicht grafische Oberflächen realisieren lassen.

18. Im Gegensatz zu anderen Frameworks, wie UPnP und DPWS, benötigen WODA-Dienste nur eine einzige Schnittstelle, um sowohl von automatischen Programmen als auch von Benutzern eines Webbrowsers verwendet werden zu können.

19. Die meisten geräteintegrierenden Frameworks besitzen das Konzept eines Gerätes, welches sich vom Konzept eines Dienstes unterscheidet. Diese Unterscheidung ist nicht notwendig, sondern erhöht nur die Komplexität des Frameworks. In WODA werden daher Geräte ausschließlich durch die Dienste identifiziert, die auf dem Gerät laufen. Die Hardware ist von untergeordneter Bedeutung.

20. Interoperabilität ist für WODA-Dienste leichter zu realisieren als für Dienste, die mit Web Services umgesetzt wurden.

Grafische Modellierungswerkzeuge und die Pipes-Plattform

21. Indem Dienste und Geräte miteinander komponiert werden (Dienstkomposition), können höherwertige Dienste bzw. neue Anwendungen geschaffen werden. Die Kombinationsmöglichkeiten steigen mit zunehmender Geräte- und Dienstvielfalt.

22. Damit diese Form der Anwendungserstellung auch für technisch weniger versierte Benutzer ermöglicht werden kann, bedarf es der Bereitstellung von Werkzeugen, die, ohne programmieren können zu müssen, funktionieren und die die Komplexität der Frameworktechnologien verbergen. Hierzu sind grafische Modellierungswerkzeuge sehr gut geeignet.

23. Die Pipes-Plattform stellt ein solches Werkzeug zur Verfügung. Mit ihr können Geräte- und Dienstkompositionen in einer intuitiven Weise modelliert und ausgeführt werden.

24. Da die Pipes-Plattform auf OSGi (Open Service Gateway Initiative) basiert, kann es für unterschiedliche Domänen, wie beispielsweise Gebäudeautomatisierung, Desktopbereich, Internet und eingebettete Systeme, verwendet werden.

25. Durch die homogene Abbildung von Funktionalitäten auf Module lassen sich framework-übergreifend Anwendungen erstellen.

26. Das Plug-in-Konzept der Pipes-Plattform erlaubt die Erweiterbarkeit durch Drittanbieter. Durch Plug-ins erweiterbare Systeme sind eine wichtige Voraussetzung für den Erfolg einer solchen Plattform.

Publikationen

Proceedings (Peer reviewed)

Andreas Bobek, Hendrik Bohn, Frank Golatowski: Voice-based Generic UPnP Control Point, 2nd IEEE International Conference on Industrial Informatics, INDIN 04, pp. 487–492, ISBN: 0-7803-8513-6, Berlin, Deutschland, Juni 2004.

Michael Ditze, Chris Loeser, Hendrik Bohn, Andreas Bobek, Frank Golatowski: Quality of Service and Proactive Content Replication in UPnP based A/V Environments, 23rd IASTED International Multi-Conference on Applied Informatics, pp. 729–734, ISBN: 0-88986-468-3, Innsbruck, Österreich, Februar 2005.

Andreas Bobek, Hendrik Bohn, Frank Golatowski: UPnP AV Architecture – Generic Interface Design and Java Implementation, 23rd IASTED International Multi-Conference on Applied Informatics, pp. 699–704, ISBN: 0-88986-468-3, Innsbruck, Österreich, Februar 2005.

Hendrik Bohn, Andreas Bobek, Frank Golatowski: Bluetooth Device Manager Connecting a Large Number of Resource-Constraint Devices in a Service-Oriented Bluetooth Network, 4th IEEE International Conference on Networking (ICN 05), pp. 430–437, ISBN: 3-540-25339-4, St. Gilles Les Bains, La Reunion, April 2005.

Andreas Bobek, Hendrik Bohn, Frank Golatowski, Gerd Kachel, Andreas Spreen: Enabling Workflow in UPnP Networks, 3rd IEEE International Conference on Industrial Informatics, INDIN 05, ISBN: 0-7803-9095-4, Perth, Australien, August 2005.

Stefan Illner, Heiko Krumm, Ingo Lück, Andre Pohl, Andreas Bobek, Hendrik Bohn, Frank Golatowski: Management of Embedded Service Systems – An Applied Approach, 20th IEEE International Conference on Advanced Information Networking and Applications (AINA 06), pp. 519–523, ISBN-ISSN: 1550-445X, 0-7695-24, Wien, Österreich, April 2006.

Hendrik Bohn, Andreas Bobek, Frank Golatowski: SIRENA – Service Infrastructure for Real-

time Embedded Networked Devices: A service oriented framework for different domains, 5th International Conference on Networking (ICN 06), ISBN: 0-7695-2552-0, Morne, Mauritius, April 2006.

Frank Reichenbach, Andreas Bobek, Philipp Hagen, Dirk Timmermann: Increasing Lifetime of Wireless Sensor Networks with Energy-Aware Role-Changing, 2nd IEEE International Workshop on Self-Managed Networks, Systems & Services (SelfMan 2006), pp. 157–170, ISBN: 978-3-540-34739-2, Dublin, Irland, Juni 2006.

Elmar Zeeb, Andreas Bobek, Hendrik Bohn, Frank Golatowski: Lessons learned from implementing the Devices Profile for Web Services, Inaugural IEEE International Conference on Digital Ecosystems and Technologies (IEEE DEST 2007), pp. 229–232, ISBN: 1-4244-0470-3, S. 229-232, Cairns, Australien, Februar 2007.

Elmar Zeeb, Andreas Bobek, Hendrik Bohn, Frank Golatowski: Service-Oriented Architectures for Embedded Systems Using Devices Profile for Web Services, 2nd International IEEE Workshop on Service Oriented Architectures in Converging Networked Environments (SOCNE 07), Niagara Falls, Kanada, Mai 2007.

Elmar Zeeb, Andreas Bobek, Hendrik Bohn, Steffen Prüter, Andre Pohl, Heiko Krumm, Ingo Lück, Frank Golatowski, Dirk Timmermann: WS4D: SOA-Toolkits making embedded systems ready for Web Services, Open Source Software and Productlines 2007 (OSSPL07), Limerick, Irland, Juni 2007.

Elmar Zeeb, Andreas Bobek, Frank Golatowski, Dirk Timmermann: Web Services – zu groß für eingebettete Systeme?, 12. Maritimes Symposium Elektrotechnik, Elektronik und Informationstechnik, pp. 179–184, Rostock, Deutschland, Oktober 2007.

Andreas Bobek, Elmar Zeeb, Frank Golatowski, Dirk Timmermann: Entwicklung standardisierter Geräte mittels Geräte- und Dienstvorlagen für das Devices Profile for Web Services, 12. Maritimes Symposium Maritime Elektrotechnik, Elektronik und Informationstechnik, pp. 185–190, Rostock, Deutschland, Oktober 2007.

Hendrik Bohn, Andreas Bobek, Frank Golatowski: Process Compiler for Resource-Constrained Embedded Systems, 3rd International IEEE Workshop on Service Oriented Architectures in

Converging Networked Environments (SOCNE 08) in conjunction with IEEE 22nd AINA2008, pp. 1387–1392, ISBN: 978-0-7695-3096-3, Ginowan, Okinawa, Japan, März 2008.

Andreas Bobek, Elmar Zeeb, Hendrik Bohn, Frank Golatowski, Dirk Timmermann: Device and Service Templates for the Devices Profile for Web Services, 6th IEEE International Conference on Industrial Informatics (INDIN 08), Daejeon, Korea, Juli 2008.

Vorträge

Andreas Bobek, Hendrik Bohn, Frank Golatowski: Voice-based Generic UPnP Control Point, 2nd IEEE International Conference on Industrial Informatics, INDIN 04, Berlin, Deutschland, Juni 2004.

Andreas Bobek, Hendrik Bohn, Frank Golatowski: UPnP AV Architecture – Generic Interface Design and Java Implementation, 23rd IASTED International Multi-Conference on Applied Informatics, Innsbruck, Österreich, Februar 2005.

Andreas Bobek, Hendrik Bohn, Frank Golatowski, Gerd Kachel, Andreas Spreen: Enabling Workflow in UPnP Networks, 3rd IEEE International Conference on Industrial Informatics, INDIN 05, Perth, Australien, August 2005.

Frank Reichenbach, Andreas Bobek, Philipp Hagen, Dirk Timmermann: Increasing Lifetime of Wireless Sensor Networks with Energy-Aware Role-Changing, 2nd IEEE International Workshop on Self-Managed Networks, Systems & Services (SelfMan 2006), Dublin, Irland, Juni 2006.

Andreas Bobek, Elmar Zeeb, Frank Golatowski, Dirk Timmermann: Entwicklung standardisierter Geräte mittels Geräte- und Dienstvorlagen für das Devices Profile for Web Services, 12. Maritimes Symposium Maritime Elektrotechnik, Elektronik und Informationstechnik, Rostock, Deutschland, Oktober 2007.